공학박사학위논문

병렬 및 분산 임베디드 시스템을 위한 모델 기반 코드 생성 프레임워크

Model-based Code Generation Framework for Parallel and Distributed Embedded Systems

2020년 2월

서울대학교 대학원
컴퓨터공학부
정 은 진

Abstract

Model-based Code Generation Framework for Parallel and Distributed Embedded Systems

EunJin Jeong
Department of Computer Science and Engineering
College of Engineering
Seoul National University

While various software development methodologies have been proposed to increase the design productivity and maintainability of software, they usually focus on the development of application software running on a single processing element, without concern about the non-functional requirements of an embedded system such as latency and resource requirements.

In this thesis, we present a model-based software development method for parallel and distributed embedded systems. An application is specified as a set of tasks that follow a set of given rules for communication and synchronization in a hierarchical fashion, independently of the hardware platform. Having such rules enables us to perform static analysis to check some software errors at compile time to reduce the verification difficulty. Platform-specific program is synthesized automatically after mapping of tasks onto processing elements is determined.

The program synthesizer is also proposed to generate codes which satisfies platform requirements for parallel and distributed embedded systems. As multiple models which can express dynamic behaviors can be depicted hierarchically, the synthesizer supports to manage multiple task graphs with a different hierarchy to run tasks with parallelism.

Also, the synthesizer shows methods of managing codes for heterogeneous platforms and generating various communication methods. The viability of the proposed software development method is verified with a real-life surveillance application that runs on six processing elements with three remote communication methods, and remote deep learning example is conducted to use heterogeneous multiprocessing components on distributed systems. Also, supporting a new platform and network requires a small effort by measuring and estimating development costs.

Since tolerance to unexpected errors is a required feature of many embedded systems, we also support an automatic fault-tolerant code generation. Fault tolerance can be applied by modifying the task graph based on the selected fault tolerance configurations, so the non-functional requirement of fault tolerance can be easily adopted by an application developer. To compare the effort of supporting fault tolerance, manual implementation of fault tolerance is performed. Also, the fault tolerance method is tested with the fault injection tool to emulate fault scenarios and inject faults randomly.

Our fault injection tool, which has used for testing our fault-tolerance method, is another work of this thesis. Emulating fault scenarios by intentionally injecting faults is commonly used to test and verify the robustness of a system. To emulate faults on an embedded system, we present a run-time fault injection framework that can inject a fault on both a kernel and application layer of Linux-based systems. For injecting faults on a kernel layer, two complementary fault injection techniques are used. One is based on Kernel GNU Debugger, and the other is using a hardware breakpoint supported by the ARM architecture. For application-level fault injection, the GDB-based fault injection method is used to inject a fault on a remote application. The viability of the proposed fault injection tool is proved by real-life experiments with an ODROID-XU4 system.

Keywords : code generation, fault tolerance, dataflow model, embedded software design methodology, fault injection, platform-aware programming, network programming,

parallel and distributed system

Student Number : 2015-30273

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The complexity of embedded systems is incessantly increasing, as can be observed in multiple areas such as mobile phones, automotive electronics systems, and intelligent robots. A high-end embedded system becomes a parallel and distributed computing system that consists of heterogeneous processing elements (PEs) interconnected with various communication methods. Application software running on such a system is a parallel and distributed program that is known to be very difficult to develop correctly. Also, an embedded system has additional requirements other than functional correctness, such as real-time requirements and satisfaction of resource constraints.

While various software development methodologies have been proposed to increase the design productivity and maintainability of software, they usually focus on the development of application software running on a single processing element, possibly many-core processors, without concern about the non-functional requirements of an embedded system. Besides, regardless of which methodology to use, the common practice of software design resorts to test or simulation to verify the functional correctness and the satisfaction of non-functional requirements. As the system complexity grows, verification by test and simulation becomes more time consuming and difficult since they are not able to consider all possible behaviors of the system.

Software development on embedded systems is different from software development on general-purpose computers. Conventional software development supposes that the hardware platform is already fixed, and its process focuses on test design to enhance software reliability. Also, performance can be easily measured by running software on real hardware. However, during embedded software development, hardware platform can be changed, and performance estimation and software verification are needed before testing on a real hardware platform. For distributed embedded system, not only a hardware platform but also a communication method can be switched because of external or internal factors such as availability of hardware or new software requirements. To overcome this challenge, some software development frameworks hide hardware and communication during software development.

There is existing software development framework which hides hardware platform from software development. AUTOSAR [1] hides hardware information by the AUTOSAR runtime environment, so software components in the application layer use AUTOSAR Interface to communicate with other applications or use hardware resources. Another software framework is ROS [2] which provides software packages to develop robot applications. It provides communication packages of a virtual communication environment which consists of each process as a node and communication between nodes with publish/subscribe mechanisms. However, these two software frameworks are targeted on domain-specific areas such as automotive or robots. Moreover, the software models of both frameworks are not using a formal model, so performance estimation and resource requirements cannot be done before testing on real hardware.

To tackle those challenges of embedded software development, a software design methodology based on formal models of computation has been proposed [3] in which an application is specified as a set of tasks that follow a set of given rules for communication and synchronization. We adopt this methodology in this work; we make computation tasks follow a dataflow model where tasks communicate with each other explicitly

through channels. The control behavior of the system is specified by a control task that is based on the FSM (finite state machine) model. By enforcing the use of formal models as much as possible, we can detect some critical errors such as deadlock possibility and buffer overflow and estimate the resource requirements through static analysis.

In this methodology, task-level software specification is performed independently of the hardware platform, unlike the common practice of software development after the hardware platform is fixed. It corresponds to the architecture design in the V-model of software development. Note that we assume that all tasks are given and already validated by unit testing. For each hardware platform, a set of tasks that contain hardware-specific codes is predefined in a task library. Since a task is a unit of mapping and scheduling at the OS level, parallelizing an application is easily performed by mapping tasks onto processors. Unlike the conventional model-driven development methodology, the initial platform-independent task-level specification is not translated into the platform-dependent specification. Instead, we generate the software code directly from the task level specification after mapping and scheduling decision is made. Automatic program synthesis from the task-level specification is the primary concern of this work.

Heterogeneous models are used for specifying an application to extend the expression capability of application behavior. Besides, hierarchically placing heterogeneous models enriches the application behavior. However, code generation from hierarchical models with dynamic behaviors is not easy for multiprocessor systems. This requires a synchronization of multiple controls from various models, so existing researches places SDF models independently [4] or hierarchically placing basic SDF models [5]. To overcome this difficulty, we provide a lock mechanism for synchronizing multiple hierarchical models, and model controller functions are used to execute model semantics in order.

After a task mapping decision is made, we synthesize a separate program that runs the mapped tasks on each processing element. Since the internal code of each task is given, two main issues in program synthesis are scheduling of the mapped tasks and

communication code synthesis between tasks. There are various ways of scheduling the mapped tasks based on the scheduling policy [6] and the OS running on the processing element. If the OS supports a multi-threaded program, we synthesize a multi-thread program by creating a thread for each task. If a task has internal parallelism, a set of threads can be created for the task. In case multi-threading is not supported in the processing element, we need to synthesize the run-time scheduling code. A multi-thread program is very likely to have concurrency bugs that are very challenging to detect and solve due to non-deterministic behavior and race condition. Many researchers [7] have struggled to detect and fix these bugs for the last decades. If the synthesized code maintains the synchronization communication semantics of the initial task-model, it can be free from such concurrency bugs.

There are two different needs of communication in the synthesized program. One is communication between two mapped tasks in the same processing element, and the other is to communicate with the other processing element. Depending on system characteristics, various kinds of external communication methods may be used. Since interface code with the outside is hard to develop and debug, there exist several approaches proposed to generate external communication code such as generating middleware [8, 9] and generating communication codes through specific languages [10, 11]. They usually assume a specific communication media or a set of communication protocols. Even with those techniques, interface code needs to be modified whenever the task mapping is changed, or the hardware component is changed. The proposed program synthesis framework provides an extensible set of communication modules to select based on the hardware components automatically, relieving the application programmer of managing the interface method.

In addition to the baseline program, we can synthesize extra code to improve the quality of the code. A good example is to increase the reliability of the application by adding extra code for fault tolerance automatically. A user of an embedded system ex-

pects the system to work correctly even when any unexpected fault occurs on a hardware component. To meet this user expectation, many researchers have proposed various software techniques [12] to increase resiliency to faults. Some examples are instruction-level replications[13, 14] and application thread replication [15, 16, 17] based on compiler modification. Manually applying those techniques to an application would be too heavy to bear for application programmers that are not familiar with the reliability issue. In the proposed methodology, extra code for fault tolerance is automatically inserted in the synthesized program, while satisfying the non-functional requirements of the embedded system.

To verify the fault tolerance of an application, we utilize a fault injection tool that can inject faults on both the kernel and application layer of the Linux system. A proposed kernel-level fault injection methods are useful to emulate a hardware fault scenarios and watch the application or system behavior. An application-level fault injection methods are proper to reproduce the same fault scenarios and random fault injection on an application.

1.2 Contribution

The main contributions of this dissertation can be summarized as follows:

- The proposed software development framework synthesizes a parallel and distributed program automatically from a dataflow model-based task-level specification of an application and a given mapping decision of tasks onto processing elements. By simply changing the task mapping, we can synthesize a new program with different performance and resource requirements. Through the static analysis of the dataflow specification, some program errors such as deadlock and buffer overflow can be detected. Also, the synthesized program is guaranteed to be free from those errors as long as the dataflow semantics is preserved in the synthesized program. It significantly reduces the burden of checking the correctness of the program by testing.

- For each processing element, we synthesize the program that includes scheduling of the mapped tasks and communication between tasks in a platform-dependent manner. In particular, the proposed framework supports various types of communication methods between processing elements, relieving the application programmer of the burden of re-writing the interface code whenever the mapping decision and hardware platform is modified.

- The proposed synthesizer generates codes from hierarchical extended SDF models to express complex dynamic behaviors working on not only a single processing element but also multiple processing elements in parallel.

- The proposed framework may add extra code to improve the quality of the program, with an example of fault-tolerant code generation in this work. Adding fault-tolerance feature is simply made at the task-level specification by utilizing the SDF/L model [18], and existent program synthesis is performed from the modified

task-level specification. Currently, it supports two fault tolerance methods: active replication and re-execution. Also, fault tolerance techniques can be selectively applied for each task, so different fault tolerance techniques or configurations can be applied to different tasks.

- The viability of the proposed software development method is verified with a non-trivial example of a distributed embedded application running on four computing devices.

- A fault injection tool for both the kernel and application layer of the Linux system is developed, and it is used for verifying the resiliency improvement of fault-tolerant programs. Also, this tool provides various fault types such as random bit-flips and hardware faults to emulate various fault scenarios.

1.3 Dissertation Organization

The rest of the dissertation is organized as follows: Chapter 2 introduces background information about HOPES [4] and UEM [3] which are the software design framework and the programming model of developing embedded software on parallel and distributed embedded systems. In Chapter 3, we present a motivation example and the proposed model-based code generation framework. Chapter 4 explains fault-tolerant code generation methods. Chapter 3 and 4 also includes related researches and corresponding experiments. In Chapter 5, a fault injection tool which is used in Chapter 4 is explained. Finally, Chapter 6 concludes our work.

Chapter 2

Background

2.1 HOPES: Hope of Parallel Embedded Software

HOPES [4] is an embedded software design framework that supports a parallel programming environment for multiprocessor embedded systems. Unlike HW/SW codesign environments, HOPES puts more emphasis on the implementation of software components. The following subsection introduces a design flow of developing embedded software in HOPES.

2.1.1 Software Development Procedure

The overall procedure of HOPES software development for multi-processor embedded systems is depicted in Fig. 2.1. An application is specified by a model-based task graph, as explained in section 2.2. Also, we prepare a set of task codes whose internal behavior is defined in a popular programming language; *C* is used in our current implementation since it is the most popular programming language for embedded devices. For platform independent specification of a task, the software development framework provides a set of application programming interfaces (APIs) to perform data communication between tasks. Those APIs are translated into platform-dependent APIs after the mapping is determined.

Given the application specification, we first perform the syntax checking of the task

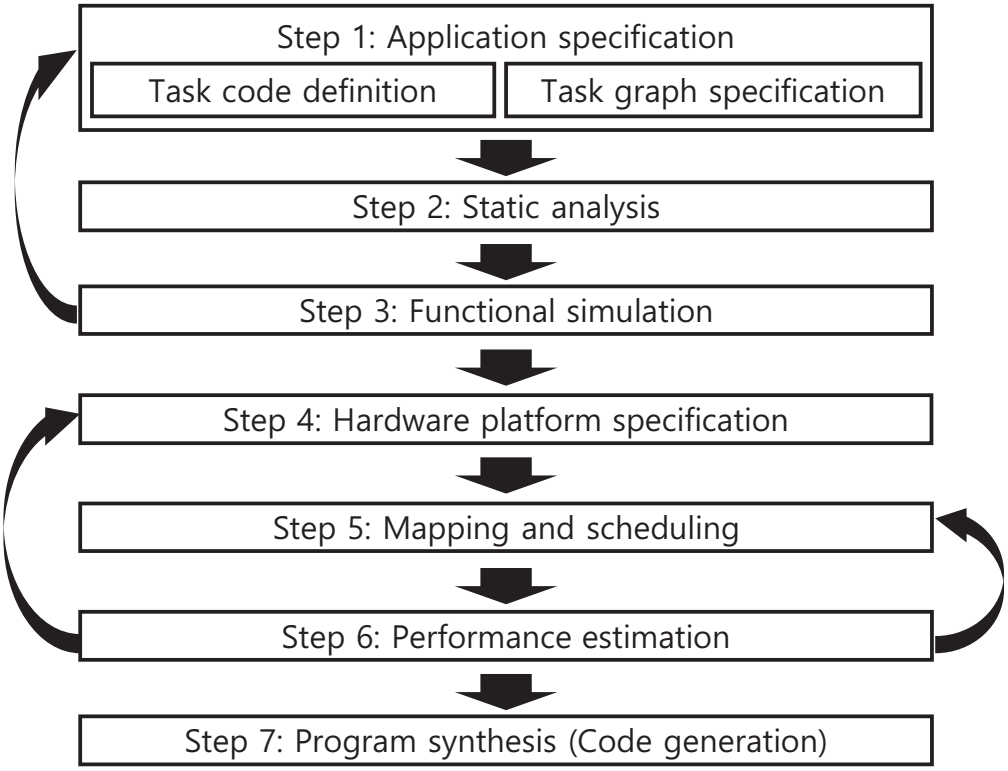


Figure 2.1: The overall procedure of model-based embedded software development

model and static analysis to check the possibility of deadlock condition and buffer overflow for each SDF subgraph. For a control task, model checking is applied to check the reachability test for a certain state. The next step is the functional simulation to check the functional correctness of the program. It is accomplished by mapping all tasks to a simulation host, synthesizing a multi-thread program, and running it on the simulation host. For a task that is dependent on a specific hardware platform, it is necessary to make a simulation model of the task for functional simulation. We repeat steps 1 to 3 until the functional correctness is verified.

After functional correctness is verified, the next step is to specify the hardware platform information that is needed in the program synthesis. The type of processing elements and the number of cores in each processing element are specified. Also, communication methods and related information need to be included.

Based on the hardware platform information, mapping of tasks is conducted manually or automatically. After a mapping decision is made, we estimate the performance and resource requirements. For performance estimation, it is necessary to know the estimated execution time of each task on the mapped processor and the resource requirement of the task. We assume that such profiling information is given a priori by unit testing. If the estimated performance does not satisfy the throughput or latency requirement of an application, we need to change the mapping decision. Among the many feasible mappings, we may want to find an optimal mapping with minimal resource requirements. This design loop of finding an optimal mapping is known as design space exploration, which makes the proposed design procedure distinguished from the conventional software development flow.

Finally, we synthesize the program that will be run on each processing element after the mapping decision is made.

For parallel and distributed embedded software development, the HOPES software development procedure can be applied without any major modifications. In detail, There

are some extra works during the steps. Because software is distributed on multiple devices, communication methods and its related information are needed to be included during the hardware platform specification step. Also, multiple synthesized programs are generated as the software is targeted on heterogeneous devices.

2.1.2 Components of HOPES

HOPES consists of three parts: HOPES User Interface (UI), UEM Programming Model, Code Generator. The HOPES user interface is an integrated design environment that can manage multiple projects of HOPES applications, so a user can draw tasks and channels to specify his or her applications. Also, it provides a simple code editor to write task codes of each task and generates a basic code skeleton to help writing codes. Furthermore, it can specify special-purpose tasks such as library tasks, loop tasks, and control tasks to show the ability to support heterogeneous models. Static analysis can be performed on the HOPES environment so that a user can analyze any problems in the task graph such as buffer overflow error, or deadlock. In addition, the hardware platform specification, mapping and scheduling decision, and performance estimation can be done on HOPES UI. Therefore, all the steps except step 7 in 2.1 are involved in this HOPES UI.

UEM Programming model is a programming model in HOPES, and it consists of application task codes, generic APIs, and meta-data information. Generic APIs are provided to communicate and control among tasks, so an application developer utilizes this API in their task codes. All the non-algorithm and task graph specifications are stored as meta-data information. The detailed models used in UEM will be introduced in the next section.

The code generator, also known as the code generation framework, is our contribution to this work. It is the final step of developing embedded software, and a detailed explanation will be shown in Chapter 3.

2.2 Universal Execution Model

2.2.1 Task Graph Specification

Figure 2.2 shows the model-based task level specification introduced in [3], which is adopted in this work. A parallel and distributed application is specified as a hierarchical task graph in which atomic tasks represent tasks running concurrently, and channels represent the data dependency between tasks. At the top level, a super task represents a group of tasks that will be mapped onto the same device or the same processing element, denoted as a virtual device in the figure.

Each virtual device contains a task graph that consists of two types of tasks, control task and dataflow task. A dataflow task is invoked, or triggered, by the arrival of data from its input ports. If the number of data samples required from each input port is fixed and not changing dynamically for each invocation, the dataflow task belongs to a specific dataflow model, called synchronous dataflow (SDF) [19]. If a task graph consists of SDF tasks, the task graph is called an SDF graph. For an SDF graph, we can check the possibility of deadlock and buffer overflow through static analysis at compile time. It is also possible to estimate the minimum resource requirements to execute the task graph by constructing a static schedule of task executions. To enjoy such benefit, it is highly recommended to define a super task that contains a subgraph of SDF tasks, as shown at the right side of the figure.

A control task depicts the dynamic behavior of the application with a finite state machine (FSM). A state is defined by the values of global variables that affect the behavior of dataflow tasks and the execution status of each dataflow task, running or waiting. If the number of possible scheduling combinations of tasks is finite, the dynamic behavior on each device can be fully specified by an FSM in a control task.

While bold lines in Fig. 2.2 represent channels for external communications between devices, the actual communication is performed by a specific task that takes in charge of

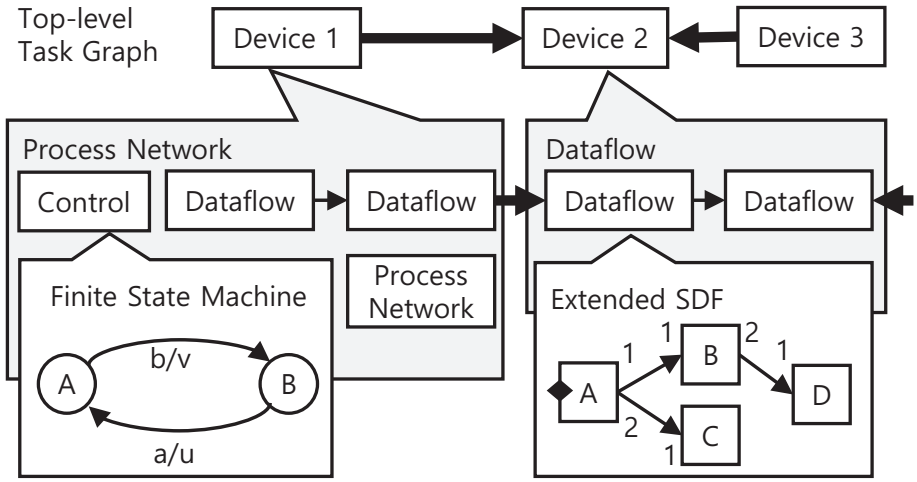


Figure 2.2: A hierarchical task graph specification of an application.

interface with the outside. A diamond shape is used as a port to express that the channel outside the task graph is connected to the specific task inside the task graph. For example, in Fig. 2.2, a diamond which is shown in task A of Device 2 means that task A receives data sent from Device 1.

Figure 3.13 illustrates the task graph specification of the surveillance application described above.

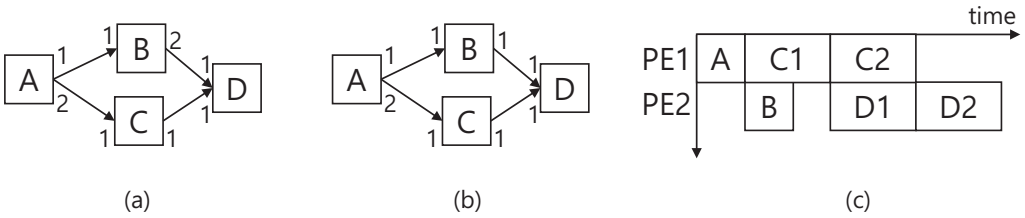


Figure 2.3: (a) An example SDF graph with annotated sample rates on the arcs, (b) an inconsistent SDF graph that has a buffer overflow error, and (c) a mapping and scheduling result of the SDF graph onto two processing elements

2.2.2 Dataflow specification of an Application

In the UEM, an application is specified by an extended synchronous dataflow (SDF) model. We first review the baseline SDF model and explain how the SDF model is extended.

2.2.2.1 Synchronous Data Flow

In the SDF model [19], an application is specified with a dataflow graph, $G(V,E)$, where V is a set of nodes and E is a set of arcs. A node $v \in V$ represents a function module, or a task, and an arc $e \in E$ is a FIFO channel between two tasks. Communication between two tasks is performed by explicit message passing via a FIFO channel. Figure 2.3 (a) shows an example SDF graph where the number annotated on the arc indicates the number of data samples, called a sample rate, to produce or consume per task execution. If unspecified, the sample rate is 1 by default. The input sample rate and the output sample rate on an arc are represented as $cons(e)$ and $prod(e)$, respectively. In the SDF model, a task becomes executable only when all input arcs have no fewer samples than the specified sample rate in the associated arcs.

By comparing the input and the output sample rates on each arc, e, we can determine the relative execution rates between the source task, denoted by $src(e)$, and the destination task, denoted by $dest(e)$. For instance, the execution rate of task C should be twice higher than that of task A in Fig. 2.3 (a), in order to make the number of samples produced from

the source task the same as the number of samples consumed by the destination task. This constraint can be formulated as the following equation, called balance equation: $prod(e) \times R(src(e)) = cons(e) \times R(dest(e))$ where $R(v)$ indicates the repetition counts of task v. An SDF graph is said to be consistent if we can find the repetition counts of all tasks to satisfy the balance equations of all arcs. Otherwise, the graph is called sample rate inconsistent, shortly inconsistent. The SDF graph shown in Fig. 2.3 (b) is inconsistent, which may incur a buffer overflow error on arc AC. An iteration of an SDF graph is defined by the set of task executions with minimum repetition counts. The minimum repetition counts of tasks in the SDF graph of Fig. 2.3 (a) are $R(A) = R(B) = 1$ and $R(C) = R(D) = 2$.

Since we can compute the minimum repetition counts of all tasks and the graph shows the dependency relationship between tasks, we can perform task scheduling at compile-time, which is to determine where and in what order tasks will be executed on a given hardware platform. By constructing a static schedule of tasks at compile-time, we can detect the critical software faults such as buffer overflow and deadlock. Figure 2.3 (c) illustrates a parallel scheduling result by mapping tasks onto two processing elements. From the parallel scheduling result, we can estimate the buffer size and the real-time performance of the graph. Note that even though there may exist numerous schedules for a given application, the determinism of the execution behavior is guaranteed meaning that the execution result is independent of the schedule.

In summary, by using the SDF model, we can verify the satisfaction of real-time requirements and resource constraints. Moreover, we can detect buffer overflow and deadlock errors at compile time. While the SDF model has the aforementioned benefits from its static analyzability, it has a severe limitation to be used as a general model for behavior specification.

Since the pure SDF model does not allow the variation of the number of data samples consumed from each input port, its expression capability is very restricted. Thus,

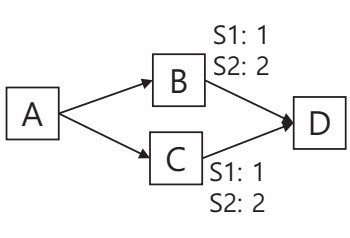
several extensions have been proposed to enhance the expression capability of the SDF model while keeping the benefit of static analyzability [18, 20, 21, 22, 23, 24]. In UEM, a dataflow task graph supports two extended SDF models, MTM (Mode Transition Machine) [25, 26] and SDF/L (SDF with Loop structure) [18] to express dynamic behavior of SDF graphs. Two SDF models will be explained in the following subsections.

2.2.2.2 Mode Transition Machine

In case an application has a finite number of different behaviors, called modes of operation, the behavior of each mode is expressed by an SDF graph, and mode transitions are specified by a tabular specification of an FSM, called Mode Transition Machine (MTM) [25]. In UEM, this model is considered as a single task with a child task graph and MTM information, and it is also called as an MTM task. It is similar to FSM-SADF [24]. An MTM describes the mode transition rules for the SDF graph, defined as a tuple Modes, Variables, Transitions where Modes and Variables represent a set of modes and a set of mode variables respectively, and Transitions is a set of transitions that consists of the current mode, a Boolean function of conditions, and the next mode. A Boolean function of transition condition is defined by a simple comparison operation between a mode variable and a value.

An example of MTM-SDF specification is shown in Fig. 2.4 in which an application has two modes of operation, S1 and S2. The input and output sample rates of a task may vary, depending on the mode. In this example, the MTM is quite simple since it needs to distinguish two modes of operation by a single mode variable. Since the granularity of a task is large and the dynamic behavior inside a task is not visible in the UEM, an MTM is not complex in general. At compile time, the SDF graph is scheduled separately for each mode of operation.

Mode transition is enabled by setting the mode variable. There are two ways of setting the mode variable. It can be set by an upper-level control task, or it can be set by a



Mode: S1, S2 Variable: cond

Transition table

Current State	Condition	Next State
S1	cond == 0	S2
S2	cond == 1	S1

Figure 2.4: Extended SDF graph with a MTM with 2 modes

designated task. A stream-based application usually starts with parsing header information that determines the mode of operation, followed by processing a stream of data. In this case, the SDF task that parses the header information is designated as a special task that may change the mode variable. In the example of Fig. 2.4, task A can be designated as a special task that determines the mode of operation.

When mode transition occurs, the SDF schedule is changed accordingly. If the mode change is enabled by the upper-level control task, it is activated at the iteration boundary of the SDF graph. If it is enabled by a designated task, mode change occurs right after the task finishes its execution. For consistency of operation, in this case, the schedules of all modes should have the same task schedule before the designated task. In case the designated task is the first task in the SDF schedule, this restriction is satisfied easily.

2.2.2.3 SDF with Loop Structure

A compute-intensive application usually spends most of its execution time in loop structures, and how to parallelize them is the main challenge for accelerating the application. Even though dataflow models, including the SDF model, are good at exploiting task-level parallelism of an application, it is difficult to exploit the parallelism of loop structures since they are not explicitly specified in existent dataflow models. In SDF, a loop structure is implicitly expressed by the same rate changes as illustrated in Fig. 2.3 (a). Among many possible schedules, a looped schedule $AB2(CD)$ can be constructed. In case 2 executions of (CD) can be parallelized with 2 output samples from A and B, a

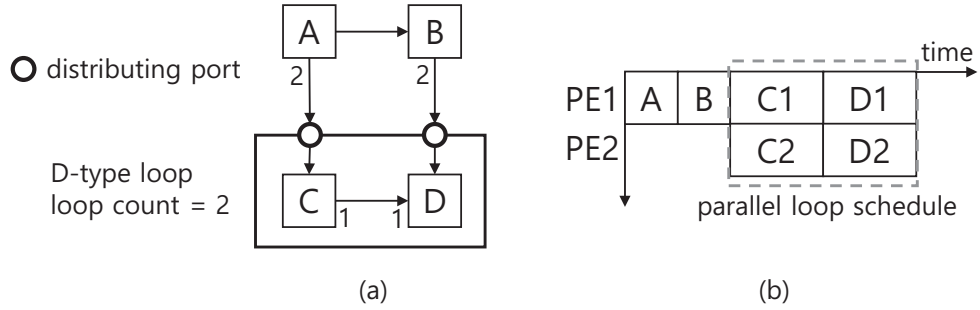


Figure 2.5: SDF graph with a loop structure

user may want to construct a parallel schedule as illustrated in Fig. 2.5 (b). But identifying such a loop structure and parallelizing it is not easy because existent parallel scheduling techniques usually aim to exploit task-level parallelism only.

Recently, we proposed a novel extension to specify a loop structure as a super node to make the SDF graph hierarchical [18]. The extended SDF graph with loop structures is called an SDF/L graph. Figure 2.5 (a) is the SDF/L graph representation of the application of Fig. 2.3 (a). In UEM, this model is considered as a single task with a child task graph and loop information, and it is also called as a loop task.

In the SDF/L model, two types of loop structures are distinguished, data loop (D-type) and convergent loop (C-type), and two types of input ports, distributing port and broadcasting port. In a D-type loop (data loop), each iteration of the loop consumes new input data from each distributing input port. The number of iterations is determined by the sample rate change of the associated input channel. The loop structure of Fig. 2.5 (a) is a D-type loop.

On the other hand, Fig. 2.6 shows an SDF graph that has a C-type loop. For a C-type loop. The C-type loop has two attributes, loop_count and exit_flag. The former is the maximum iteration count, and the second is set by a designated task, task *C* in this figure. The number of iteration is dynamically decided by the result of a computation that will set the exit_flag. All input ports of a C-type loop should be broadcasting ports from which input samples are reused in all iterations of the loop; the sample rate of the output

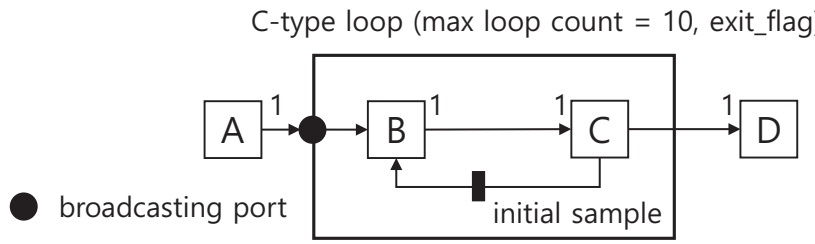


Figure 2.6: SDF graph with a C-type loop structure

connection is equal to the sample rate of the input connection.

In summary, the SDF model in UEM is extended to express dynamic behavior with an MTM, to allow the use of shared resources with a library task, and to explicitly specify the loop structures hierarchically. Refer to the corresponding references for a detailed explanation of each extension. Note that these extensions preserve the static analyzability of the SDF model. We perform static scheduling for each mode of operation. In the SDF/L model, static scheduling can be performed hierarchically from the bottom layer. A loop structure is encapsulated as a regular SDF task at the upper layer.

In this thesis, SDF/L is used for not only expressing iterative behavior of applications but also applying fault-tolerant techniques to each task, and the explanation is described in Chapter 4.

2.2.3 Task Code Specification and Generic APIs

To write a task code in HOPES, an application developer must specify three functions: TASK_INIT, TASK_GO, and TASK_WRAPUP. The TASK_INIT function is executed when the task is initialized, and the TASK_WRAPUP function is executed when the task is going to be stopped. The TASK_GO function is the main body that will repeat until the task is terminated. Inside these functions, the generic APIs listed in Table 2.1 can be used for developing an application. A prefix started with *UFPort* indicates APIs related to accessing task ports, so it allows a task to communicate with other tasks. Two kinds of read and write functions are provided depending on how to read or write data. A buffer-type read operation reads data from a channel but does not remove data in a channel. On the other hand, a queue-type read operation removes read data in a channel, and it blocks when data is not prepared yet. A prefix started with *UFTask* is used for accessing information about a task so that a user can get task parameter values or state information. Functions started with a prefix *UFControl* is related to controlling tasks, so these functions are only allowed to be used by a control task.

Table 2.1: List of generic APIs provided by UEM

API Name	Detail
UFPort_Initialize	Initialize a task port
UFPort_ReadFromQueue	Read data from queue. Read data is removed.
UFPort_ReadFromBuffer	Read data from buffer. Read data is remained.
UFPort_WriteToQueue	Write data to the channel as a queue.
UFPort_WriteToBuffer	Write data to the channel as a buffer.
UFPort_GetNumOfAvailableData	Check the number of data can be read from channel.
UFPort_Finalize	Finalize a task port.
UFPort_GetChannelSize	Get the buffer size of the channel
UFTask_GetIntegerParameter	Get an integer-type task parameter.
UFTask_SetIntegerParameter	Set an integer-type task parameter.
UFTask_GetFloatParameter	Get a float-type task parameter
UFTask_SetFloatParameter	Set a float-type task parameter
UFTask_GetState	Get task state
UFTask_GetCurrentModeName	(MTM only) Get current mode name
UFTask_SetModeIntegerParameter	(MTM only) Set integer-type mode parameter.
UFTask_UpdateMode	(MTM only) Update mode state.
UFTimer_Set	Set timer.
UFTimer_GetAlarmed	Get an alarm from a timer.
UFTimer_Reset	Reset a timer.
UFControl_RunTask	(Control task only) Run a task.
UFControl_StopTask	(Control task only) Stop a task.
UFControl_SuspendTask	(Control task only) Suspend a task.
UFControl_ResumeTask	(Control task only) Resume a task.
UFControl_CallTask	(Control task only) Call a task.
UFLoop_GetIteration	(SDF/L only) Get current loop iteration number.
UFLoop_StopNextIteration	(SDF/L convergent-type only) Stop a loop at next iteration.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <CIC_Architecture target="aaa" xmlns="http://peace.snu.ac.kr/
       CICXMLSchema">
3    <elementTypes>
4      <elementType subcategory="CPU" sleepPower="80000"
           scheduler="RR" relativeCost="1.0" name="i7_0" model="
           i7" clock="3400" category="processor" activePower
           ="80000" OS="LINUX"/>
5      <elementType name="SHARED_MEMORY_0" category="memory">
6        <slavePort size="8000" name="Slave" metric="MiB"/>
7      </elementType>
8    </elementTypes>
9    <devices>
10     <device runtime="native" platform="linux" name="aaa"
           architecture="x86_64">
11       <elements>
12         <element type="i7_0" poolSize="4" name="i7_0"/>
13         <element type="SHARED_MEMORY_0" name="SHARED_MEMORY_0
               "/>
14       </elements>
15       <connections/>
16       <modules/>
17       <environmentVariables/>
18     </device>
19   </devices>
20   <connections/>
21 </CIC_Architecture>
```

Figure 2.7: The example of meta-data information describing hardware platform

2.2.4 Meta-data Specification

Meta-data information is basically inserted in HOPES UI, but the internal data is stored as XML files. There are five types of XML files: algorithm, architecture, mapping, schedule, and configuration. These files are used as input to the code generator. The algorithm XML file stores a task graph information, so it contains all the information drawn in a task graph diagram. The architecture XML file contains a list of devices and its internal processors. Also, connectivity information is also included. Mapping information contains which task mapped to a processor, and the schedule file provides a scheduling order of each processor if a schedule is generated during mapping and scheduling step in 2.1. Finally, the configuration file contains a time to be executed and execution policy.

Chapter 3

Program Synthesis for Parallel and Distributed Embedded Systems

3.1 Motivational Example

We present a surveillance application as a motivational example of a parallel and distributed embedded system. Figure 3.1 shows the system configuration that consists of four computing devices. The surveillance system monitors strangers through a smart camera that runs an objection detection algorithm inside. If a stranger is detected, security robot 1 moves to the location where the camera finds the stranger. If the camera no longer detects a person, the robot will return to its original position. Also, there is another security robot, called security robot 2 that can be controlled remotely by the administrator. In the autonomous mode of operation, security robot 2 runs an object tracking algorithm to follow security robot 1. Other cooperation mission can be defined with security robots, and more robots can be added. Robots also have their own camera modules. All scenes captured from all devices are monitored through the monitoring console, and the console encodes and stores the captured images for recording.

The surveillance system requires multiple devices to communicate over the network. It includes intra-PE communication inside each device to receive images from the camera sensor and inter-PE communication to send captured images or control the robots. Be-

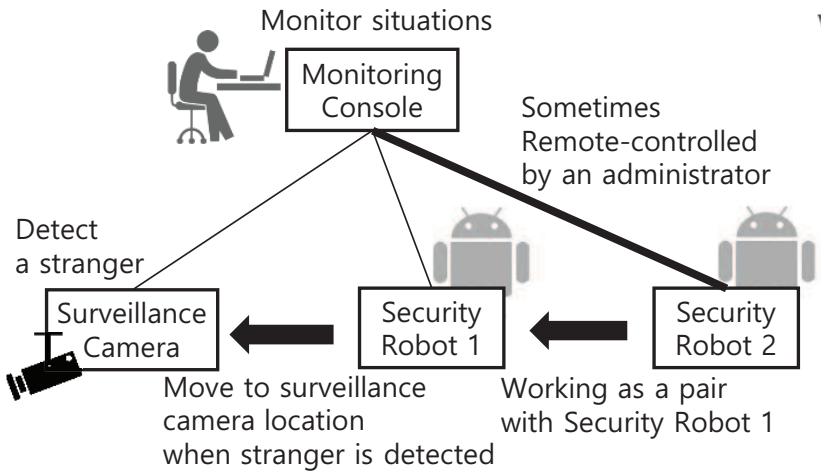


Figure 3.1: Surveillance system as a motivational example

sides, there are many computation-intensive tasks involved in this system such as object detection, image encoding, and tracking the robot. Therefore, this non-trivial application requires diverse types of connectivity for remote communication between devices and the exploitation of parallelism of computation-intensive applications. We will specify this application by the task-level specification model which is explained in section 2.2. The synthesized program from the task-level specification is discussed in section 3.5.

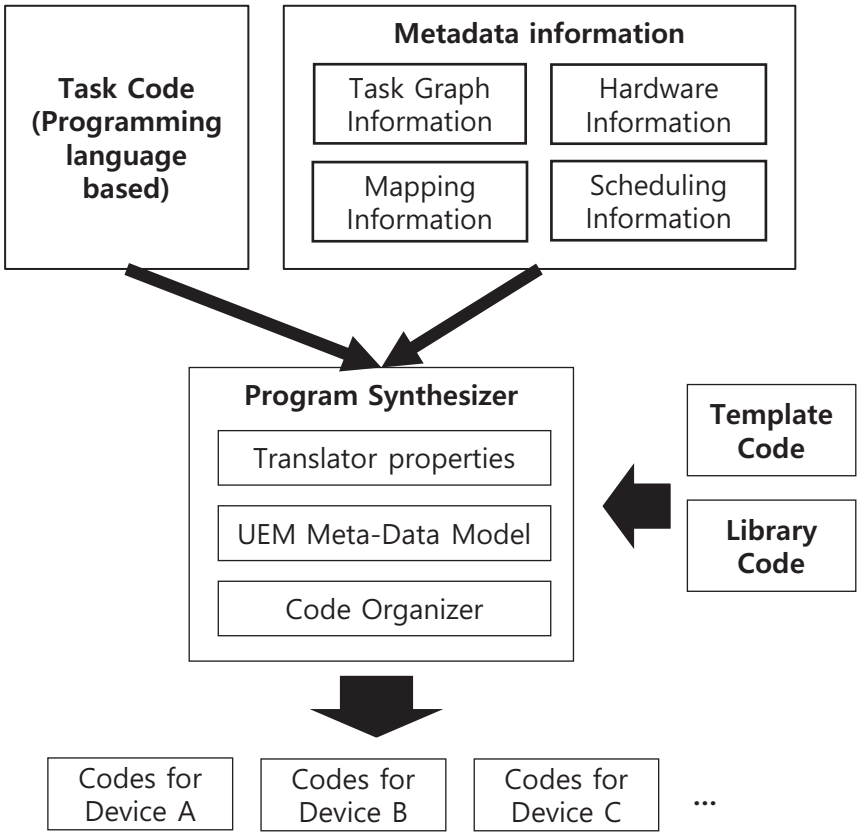


Figure 3.2: The structure of program synthesizer

3.2 Program Synthesis Overview

The program synthesizer generates target codes for each device from the UEM programming model. Fig. 3.2 describes the structure of the program synthesizer. A task code set and meta-data information are used as an input of the program synthesizer, and meta-data information is filled during step 1 to 6 in Fig. 2.1. The other input is template codes and library codes, which are described in the translator properties. The template codes contain skeleton codes and tags to generate codes from meta-data information, and the library codes consist of pure codes which are included as a file to build a target code. Overall information of template codes and library codes is shown in translator properties. By changing two code sets and translator properties, the program synthesizer has

the potential to support different programming languages or target-dependent execution models. The program synthesizer utilizes meta-data information to make the UEM meta-data model which is used for filling data to the template codes, and the code organizer determines which files from the library codes to be used for each target device.

The skeleton of the synthesized program for a device displayed in Fig. 3.3 with a simple example where three tasks are mapped to two devices. The program synthesizer synthesizes a program that will run on each device, based on the input information of software specification, hardware platform specification, and mapping decision. Two platform-specific code sets are provided for program synthesis: one is the library code, and the other is the template code. The library codes are written in a target code language and added to the target application software as a file itself. The template codes are written in a template language, and it is used for generating code segments that depend on the input information, structured task data and structured channel data, as illustrated in the figure inside light orange boxes. The main program first initializes the channel data structure using the channel library code. Next, it registers the mapped tasks to the task manager and runs the registered tasks, using the task library code. As denoted with dashed green lines, there are two ways of running the tasks. In case multi-thread scheduling is supported by the OS, it creates a thread for each task; a separate thread, *MappedThread*, is created for each task based on the given task code. If there is no pre-installed thread scheduler in the device, the program synthesizer creates a task scheduler, *ScheduledThread*. Three functions called in the *ScheduledThread* routine are also generated from the template code. In addition to the program, the program synthesizer creates a build file that is tailored for the device.

In the program synthesis flow, we focus on the communication code synthesis and the fault-tolerant code synthesis in this thesis. To support distributed embedded systems, we aim to make the communication code synthesis technique extensible and flexible to support various communication methods between devices. Making a task code fault-

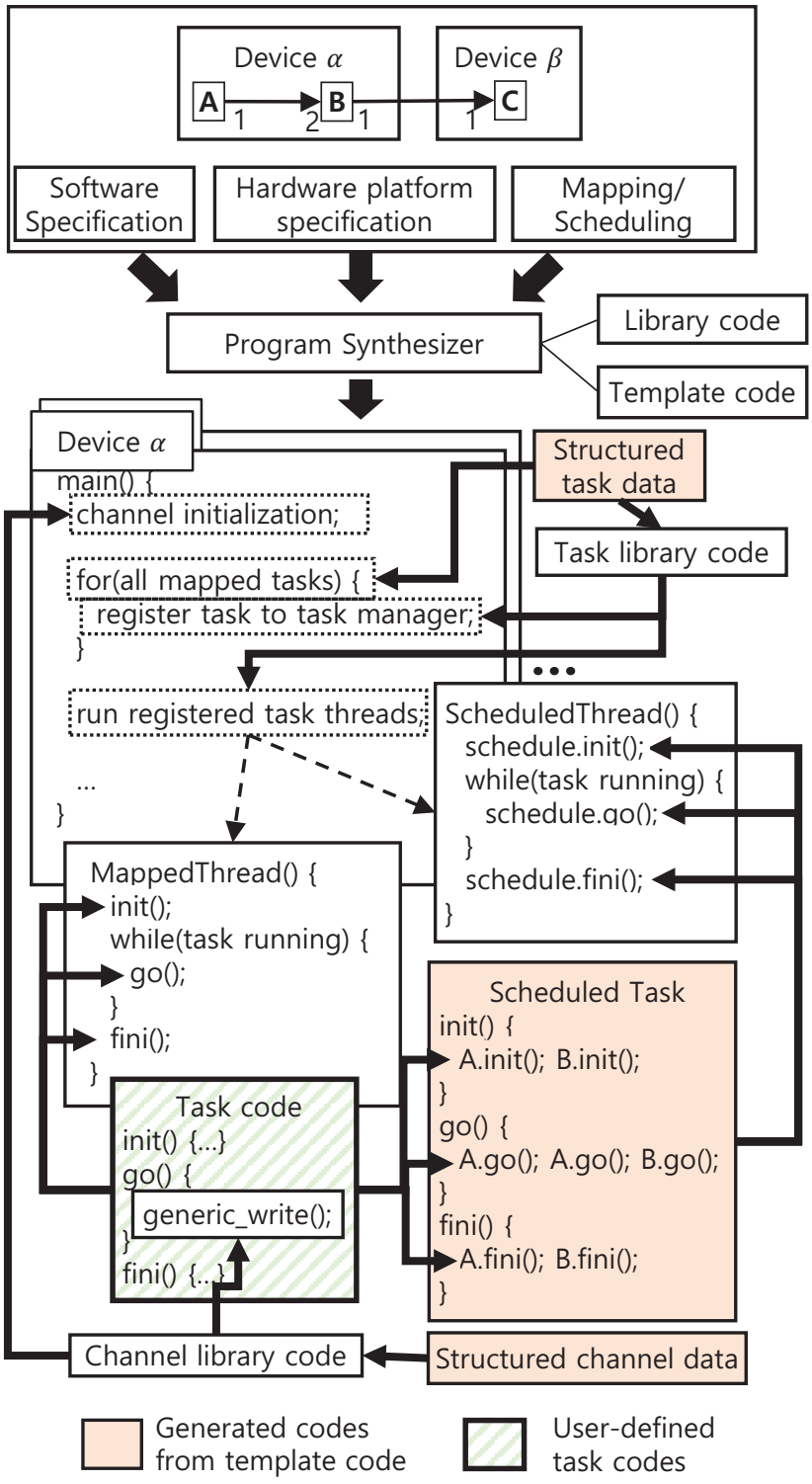


Figure 3.3: The skeleton of an example synthesized program

tolerant is achieved by modifying the task graph in the proposed method. These two techniques are explained in section 3.4 and 3.5.

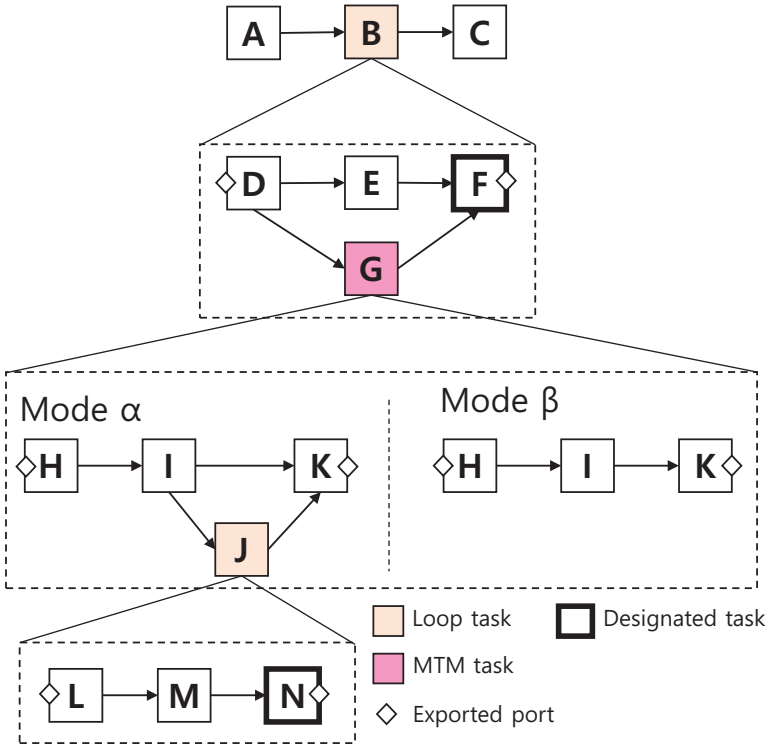


Figure 3.4: Example of hierarchically-mixed extended SDF models

3.3 Program Synthesis from Hierarchically-mixed Models

Hierarchically mixing different types of dataflow graphs can improve expression capability of dynamic behavior. Since MTM tasks can be considered as a single-level *if/switch* statement, hierarchical MTM tasks can be act as multi-level *if/switch* statements. In addition, convergent-type loop tasks can be act as *for*-statement with *break* condition. Figure 3.4 shows the example of Hierarchically-mixed loop and MTM tasks. It consists of two loop tasks and one MTM task. All loop tasks in Fig. 3.4 are convergent-type, so they have a designated task which can terminate the loop before reaching the maximum loop iteration. The loop task *J* is conditionally executed depending on parent MTM tasks' mode. Also, the loop task *G* can affect the execution of the MTM task *G*. Hierarchically-mixed models can be described with the complex pseudo-code of dynamic behaviors with

```
j J_Loop(i2)
{
  for(i = 0 ; i < 3 ; i++) {
    l = L(i2);
    m = M(l);
    n, exit = N(m);
    if(exit == true)
      break;
  }
  return n;
}

g G_MTM(d)
{
  h, mode = H(d);
  if (mode α) {
    i1, i2  = I(h);
    j = J_Loop(i2);
    k = K(i1, j);
  }
  else { // mode β
    i1  = I(h);
    k = K(i1);
  }
  return k;
}

b B_Loop(a)
{
  for(i = 0 ; i < 5 ; i++) {
    d = D(a);
    e = E(d);
    g = G_MTM(d);
    f, exit = F(e, g);
    if(exit == true)
      break;
  }
  return f;
}

a = A();
b = B_Loop(a);
C(b);
// end of single iteration
```

Figure 3.5: Pseudo-code representation of the hierarchical mixed model in Fig. 3.4

multiple functions. Figure 3.5 is an pseudo-code representation of Fig. 3.4. Functions started with capital letter are the tasks in Fig. 3.4, and lower-case variables are used as arguments and return values. Note that each hierarchical task graph is defined as a function with multiple statements calling other tasks. A task graph is also considered as a task of a upper task graph. To transfer data across task graphs with different hierarchy, we defined a exported port. We suppose that all the exported input ports are already contains enough data to execute a task graph, so blocking is not occurred by exported ports. These exported ports are used as arguments and return values of the task graph functions in Fig. 3.5.

The program synthesizer can generate codes from complex concurrently-running hierarchical models. Code generation for a single processing element from multi-level models is relatively simple than that for multiple processing elements, and Fig. 3.5 is a good example of generating codes for a uni-core system. On the other hand, code generation for multiprocessor systems must consider the parallelism, and multiple tasks can be

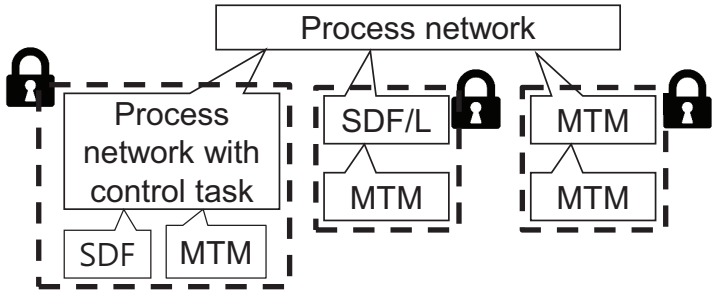


Figure 3.6: Task graph lock protection mechanism

run in parallel like Task *E* and child tasks in *G*. Also, MTM tasks or convergent-type loop tasks contain a dynamic behavior. If these task graphs are placed hierarchically without consideration, task control can be violated by the tasks in the upper or lower task graph. Furthermore, a control task with FSM can run or stop dataflow graphs, so a program synthesis tool must consider these conflicts of task control.

To support code generation on multiprocessor systems from hierarchically-mixed extended SDF models, we defined a task graph lock for synchronization among tasks that share the same ancestor. Pessimistically, this concept may result in a single global lock of all task graphs. To avoid the problem, our code synthesis tool creates a lock on only three cases: SDF/L, MTM, and a process network containing a control task. Figure 3.6 shows the example of avoiding a global lock. If a control task is not controlling some task graphs, those task graphs can be drawn separately from the upper task graph with a process network. The example figure shows that two hierarchical task graphs are separately managed from the task graph with a control task. Another consideration point for managing hierarchical task graphs with dynamic behavior is a control order of parent and child task graphs. In this case, we defined a model controller function for each task graph with dynamic behaviors, and call each model controller function from the leaf to the root ancestor. Because a parent task graph has higher priority, a lower-level task graph decision can be overridden by the upper-level task graph.

3.4 Platform Code Synthesis

Since the synthesized programs are running on heterogeneous devices, the program synthesizer needs to choose and organize proper codes for each device with different platforms. The management of code for multiple platforms is somewhat painful because some codes can be shared with multiple platforms, and some of the codes are fully target-dependent. To resolve this problem, we classify hardware platforms into three types: *constrained*, *unconstrained*, and *semi-constrained*. *Unconstrained* type is considered as conventional operating systems such as Linux, Unix, Windows, Mac, and Android (also known as LUWMA), so OS-supported APIs and dynamic memory allocation are allowed for programming. On this platform type, target hardware has enough memory to tolerate the amount of memory usage from the generated codes except the code segments developed and determined by application developers. *Constrained* type is classified as a micro-controller with no operating system, and only single-core programming is possible. Also, this type of platform has a very small amount of memory, so generated codes such as structured task data and channel data in Fig. 3.3 can be a burden for developing applications. For this reason, limited generic APIs and dataflow models are supported for this platform type. The last platform type is *semi-constrained*, and this could be placed between *constrained* and *unconstrained*. Currently, supporting this hardware platform type is left as future work.

Based on hardware platform classification, codes are managed into three categories. The first category is target-independent codes. Codes in this family are shared by all platforms. The second one is Loosely target-dependent codes, and codes in this category are shared by the same platform types mentioned in the previous paragraph. The Last one is a tightly target-dependent code set, and codes in this category can only run on a specific platform such as Arduino, Linux, and Windows. Fig. 3.7 shows which part of the codes belongs to each category, and it also depicts the layered structure of the synthesized program. The higher layer of the codes can be accessed to the functions from the lower

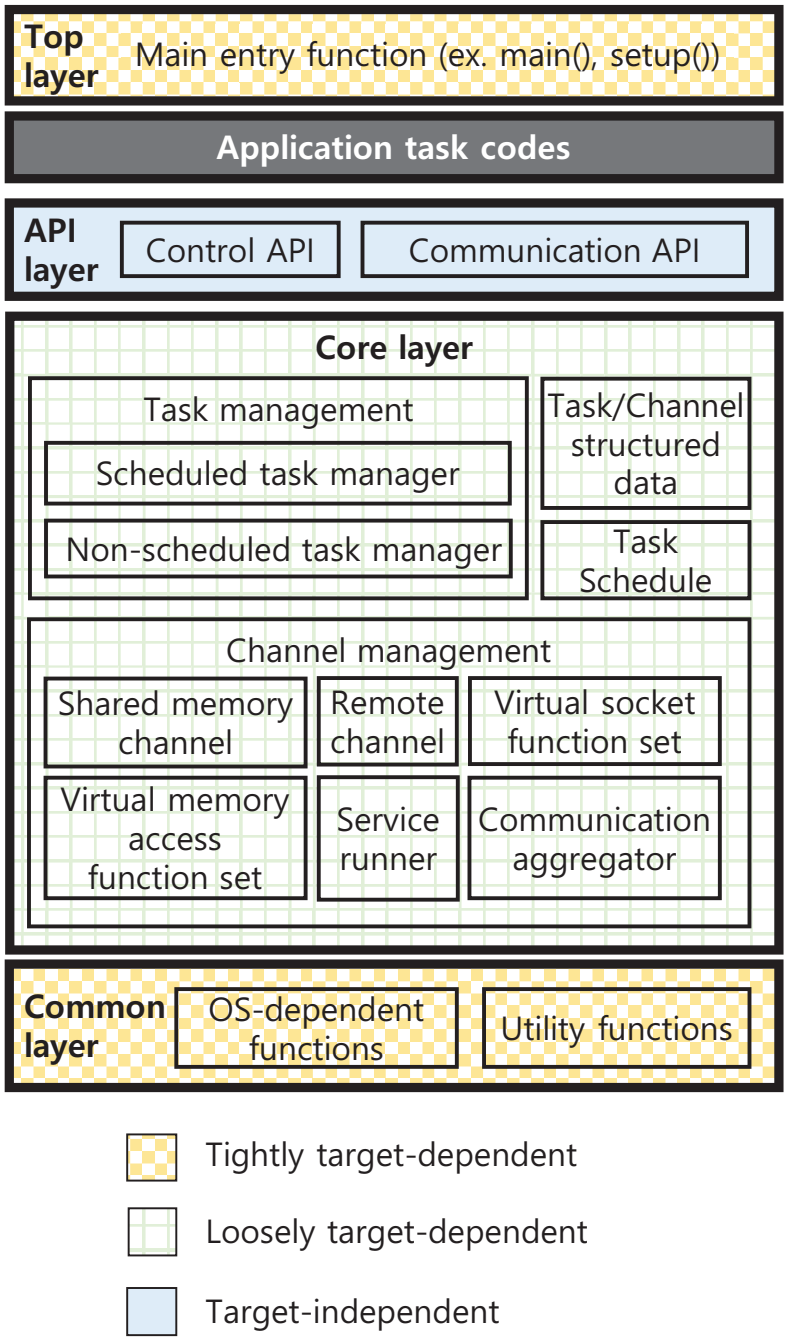


Figure 3.7: Code management structure of the program synthesizer

layer, so the top layer can access to all the lower layer functions. In the top layer, the entry point of the binary is located. For example, the *main()* function which is the entry of the program from most conventional operating systems is located here. Otherwise, for the Arduino platform, *setup()* and *loop()* functions are placed on this layer. The API layer provides an interface to the application task codes as general APIs, so channel communication and task control APIs are located in this layer. The core layer contains the main logic of executing tasks and accessing channels. The reason why this part is loosely target-dependent is that constrained and unconstrained target category has a huge difference in implementation such as multi-threading or dynamic memory allocation. The last layer is the common layer which wraps the OS-dependent APIs to use on the higher layer and also provides utility functions that can be utilized by upper layers.

The program synthesizer needs to generate codes for each target device with minimized information to reduce resource usage. First, it only includes tasks and channels which belong to the mapped device. This data management is important because constrained devices only have a few kilobytes of memory, so structured task and channel data can be a burden to run a program. Also, the synthesized program not always needs all codes from each layer, so the synthesizer excludes the library codes which are not used. For example, if an application does not use GPU, GPU-related codes can be omitted for code generation. To manage optional codes, we defined a term called *peripheral* which can be selectively added to the generated program depending on the hardware platform specification.

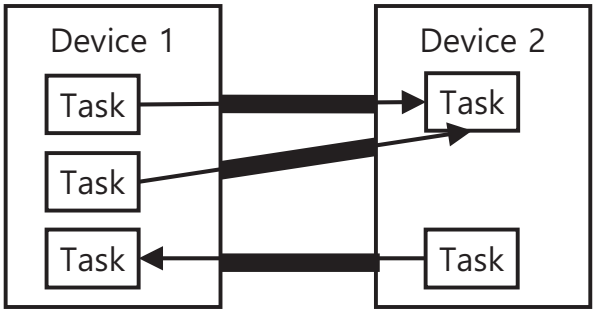
3.5 Communication Code Synthesis

Since channels that are used for data communication between tasks can be accessed by an application programming interface (API) in a task, an application programmer uses the API without knowledge of how tasks are communicated with each other during task code development. Also, in the task graph specification phase, the programmer does not know which processing elements the tasks are mapped to and which communication methods will be used between devices. For each communication API, the platform-dependent communication code is synthesized after the mapping decision is made. If two tasks are mapped to the same processing element, channel communication is likely to be accomplished by shared memory accesses. Otherwise, we need to synthesize the remote communication code that depends on which communication method is used.

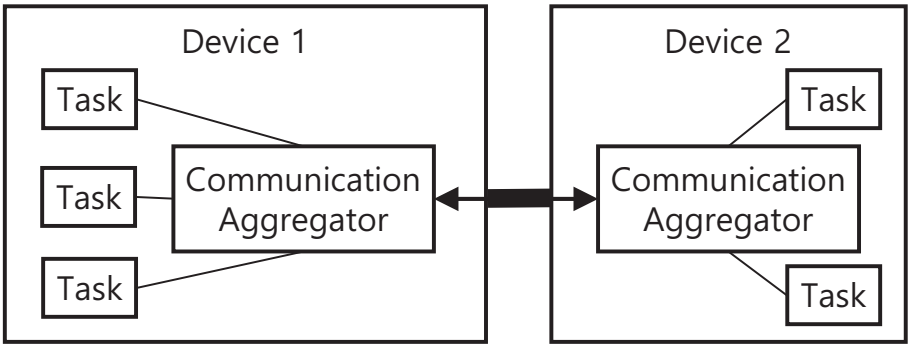
Figure 3.8 shows two different connection types for an example task graph where five tasks are mapped to two devices, and bold lines represent remote communication between devices. In the first example of Fig. 3.8 (a), a separate connection is made for each channel. For example, TCP communication allows multiple connections so that each channel may establish a separate connection between devices. On the other hand, Bluetooth communication allows only a single connection between two devices, so multiple channel communications between tasks need to be serialized. Fig. 3.8 (b) shows the case in which a single physical connection is established between devices, and logical channels are merged and serialized in the communication aggregator in each device.

Our code generation framework provides multiple modules to handle various communication methods and types. The list of the modules is shown in Table 3.1. Each module is selectively used depending on communication methods and connection types.

Figure 3.9 shows how a generic API is translated to the actual platform-dependent communication code depending on the communication type. For local communication, only a memory access function in the virtual memory access function module is called as shown in Fig. 3.9 (a). In the case of remote communication with a TCP connection,



(a) An individual connection example



(b) An aggregate connection example

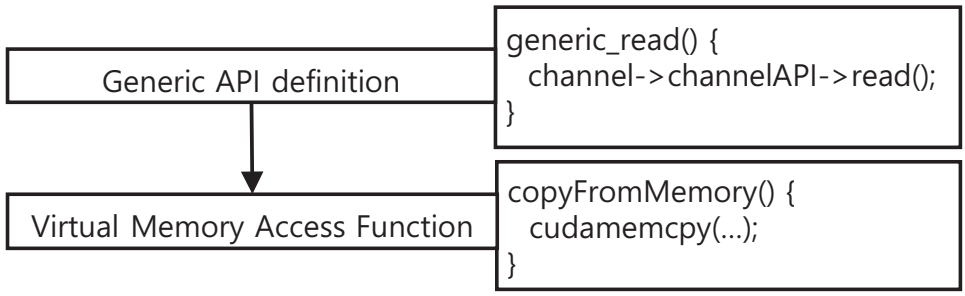
Figure 3.8: Examples of connection types between two devices

Table 3.1: Modules used for communication code generation

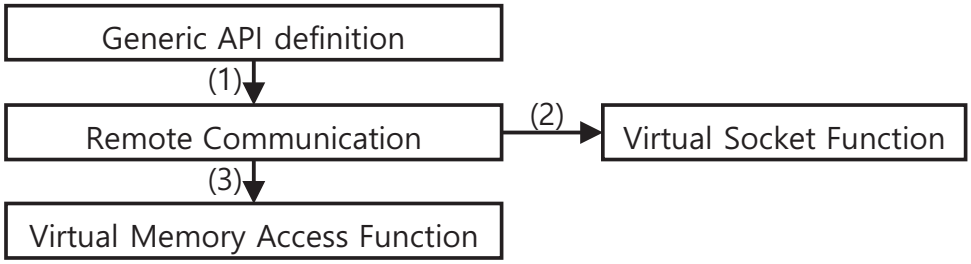
Module	Explanation
Generic API Definition	Top-level definitions of generic APIs based on communication types.
Remote communication	A set of remote communication APIs common to all communication types.
Virtual memory access function	A set of memory access functions that are dependent on the hardware platform
Virtual socket function	A set of remote communication functions specific to the communication methods
Communication aggregator	Standalone module that serializes and manages data transfer from multiple channels for remote communication

four modules are involved in the synthesized communication code; Fig. 3.9 (b) shows the calling order among the modules. Once the remote communication is established, a function in the virtual socket function module is called for actual communication. The read data from the outside is written to the local memory by a function in the virtual memory access function module. Remote communication with aggregated connection uses all five modules as displayed in Fig. 3.9 (c). The communication aggregator module is a standalone thread that collects communication requests from all communicating tasks through inter-thread communication and serializes and sends them with a function in the virtual socket function module.

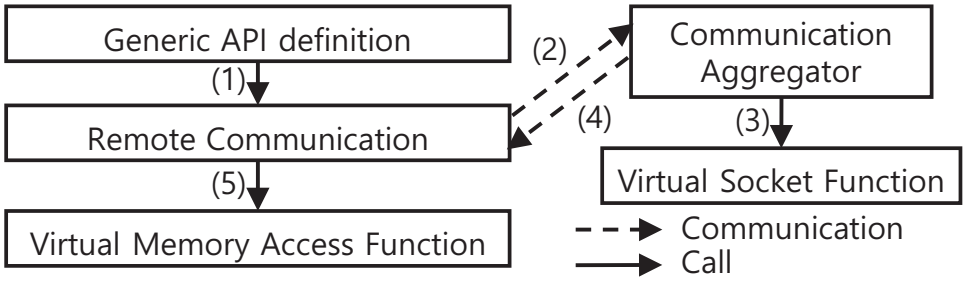
The virtual socket function module defines 8 abstract functions: *create()*, *destroy()*, *listen()*, *accept()*, *connect()*, *disconnect()*, *send()*, and *receive()*. Figure 3.10 shows various implementations of an abstract function, *connect()*, according to the communication method. Although Bluetooth, TCP, and serial communication all show differences in their implementation, the remote communication module or the communication aggregator can establish a connection by calling *connect()* function. Besides, we may support a new communication type by adding the associated implementation.



(a) Channel for local communication



(b) Channel for remote communication (individual connection)



(c) Channel for remote communication (aggregated connection)

Figure 3.9: Channel communication module for each communication type

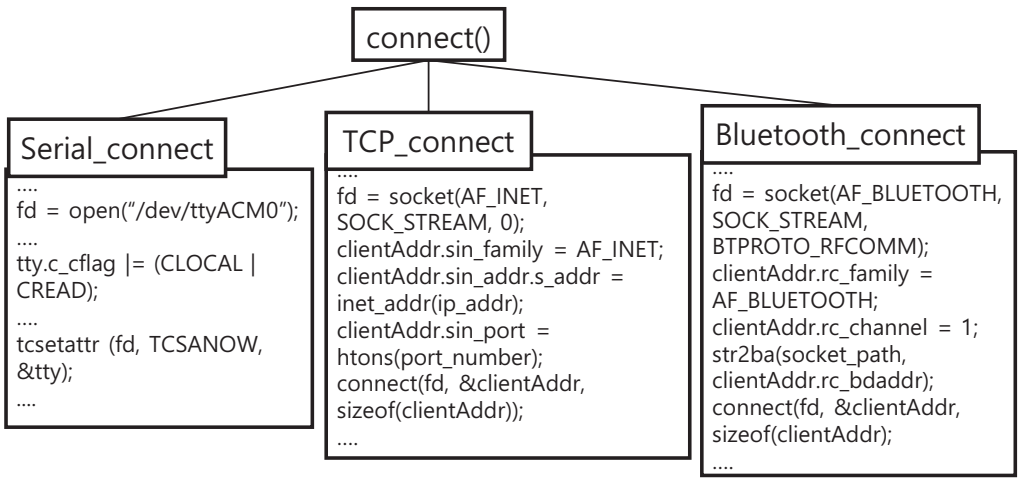


Figure 3.10: An example of a virtual socket function connect() for each communication method

3.6 Experiments

3.6.1 Development Cost of Supporting New Platforms and Networks

As our program synthesis tool is targeted on distributed and parallel embedded systems, the extension of supporting new platforms or communications are needed. Our program synthesis framework is considered to minimize the efforts of new platforms or networks. To prove this merit, we analyzed the number of lines used for generating programs. As mentioned in Fig. 3.7, code management structure consists of 4 layers except for application task codes. The number of lines per each layer is shown in Table 3.2. The critical part of supporting new platform variants is related to the top and common layers. According to the table, tightly platform-dependent codes which are classified by platform names such as Linux and Arduino are small compared to other types of codes. The codes in the tightly target-dependent layers include communication codes and GPU codes, so the primary codes for supporting a new platform is limited to task execution and synchronization logics such as threads, locks, and event handling. Also, header files are shared among platforms with the same categories, so only definition codes are needed

Table 3.2: The number of target code lines per each layer used for program synthesis

Layers			C code	C++ Code	Header
API Layer			726		215
Core Layer	Common		838		338
	Constrained		2,213		305
	Unconstrained		11,547		1,244
Common Layer	Common		310		237
	Constrained	Common			57
		Arduino	32	116	
	Unconstrained	Common	720		
		Linux	2,033		
Top Layer	Arduino			68	
	Linux		202		
Total			18,621	184	2,733

to be implemented for newly supported platforms. For example, in Fig. 3.11, a declaration of function *UCThreadEvent_SetEvent* is shared, but the definition of each platform is different. Fig. 3.11 (a) uses a pthread conditional variable and pthread mutex lock to implement the *UCThreadEvent_SetEvent* function, but Fig. 3.11 (b) uses a native Windows SDK function, *SetEvent*, for implementation on Windows.

We computed the number of codes which is related to support a new platform, and we extend our supported platforms to measure how much load is needed for implementation. Figure 3.3 shows the number of lines for providing a new platform. Currently, Linux is now supported as an unconstrained target, and its related codes are about 859 lines including both the common and top layer. Linux target codes are highly portable to other Unix/BSD-based operating systems, and automake [27] is used for making a portable Makefile which is commonly used on POSIX platforms. Also, most of functions are implemented with POSIX APIs, so the efforts of supporting a new POSIX-supported operating system is very low. According to the table, it only needs 10 more lines, and it indicates the codes related to processor mapping logic. For supporting a non-POSIX operating system, we implemented Windows-part codes to support Windows platform, and it only requires 495 lines even though the APIs are mostly different to POSIX or

```
uem_result UCThreadEvent_SetEvent(HThreadEvent hEvent) {
  uem_result result = ERR_UEM_UNKNOWN;
  SThreadEvent *pstEvent = NULL;
  uem_bool bMutexFailed = FALSE;
  int nErrorNum = 0;
#ifdef ARGUMENT_CHECK
  if (IS_VALID_HANDLE(hEvent, ID_UEM_THREAD_EVENT) == FALSE) {
    ERRASSIGNGOTO(result, ERR_UEM_INVALID_HANDLE, _EXIT);
  }
#endif
  pstEvent = (SThreadEvent *) hEvent;

  if (pthread_mutex_lock(&(pstEvent->hMutex)) != 0) {
    bMutexFailed = TRUE;
  }
  pstEvent->bIsSet = TRUE; // event is set
  // send a signal
  nErrorNum = pthread_cond_broadcast(&(pstEvent->hCond));

  if (bMutexFailed == FALSE &&
    pthread_mutex_unlock(&(pstEvent->hMutex)) != 0) {
  // ignore error
  }
  if(nErrorNum != 0) {
    ERRASSIGNGOTO(result, ERR_UEM_INTERNAL_FAIL, _EXIT);
  }

  result = ERR_UEM_NOERROR;
_EXIT:
  return result;
}
```

(a) Linux code implementation of *UCThreadEvent_SetEvent*

```
uem_result UCThreadEvent_SetEvent(HThreadEvent hEvent) {
    uem_result result = ERR_UEM_UNKNOWN;
    SThreadEvent *pstEvent = NULL;
    BOOL bSuccess;
#ifdef ARGUMENT_CHECK
    if (IS_VALID_HANDLE(hEvent, ID_UEM_THREAD_EVENT) == FALSE) {
        ERRASSIGNGOTO(result, ERR_UEM_INVALID_HANDLE, _EXIT);
    }
#endif
    pstEvent = (SThreadEvent *) hEvent;

    bSuccess = SetEvent(pstEvent->hEvent);
    if(bSuccess == FALSE) {
        ERRASSIGNGOTO(result, ERR_UEM_INTERNAL_FAIL, _EXIT);
    }

    result = ERR_UEM_NOERROR;
_EXIT:
    return result;
}
```

(b) Windows code implementation of *UCThreadEvent_SetEvent*

Figure 3.11: Target-dependent code implementation example in the common layer

Table 3.3: The number of lines for supporting new platforms

Target	Common layer	Top layer
Unconstrained (Linux)	657 lines	202 lines
Unconstrained (new POSIX-supported OS)	10 lines	None
Unconstrained (new non-POSIX OS)	462 lines	33 lines
Constrained (Arduino)	70 lines	68 lines

Table 3.4: The number of lines for supporting new communication methods

Communication method	Common layer	Core layer
TCP (Linux target)	109 lines + 568 lines	109 lines
Serial (Linux target)	309 lines	169 lines
Bluetooth (Linux target)	109 lines + 568 lines	91 lines
Bluetooth/Serial (Arduino)	89 lines	667 lines

Linux-based APIs. For developing software on a microcontroller such as Arduino, The portion of target-dependent codes are very small because it does not contain multi-tasking contents such as threads and locks.

Communication codes are located in the common and core layer, so new codes for a new communication method need to be changed on both layers. While the core layer communication codes are used to implement a virtual socket function set mentioned in Table 3.1, codes in the common layer actually use platform-dependent functions such as *recv*, *socket*, and *fopen*. Table 3.4 represents the number of codes for each communication method. For supporting TCP, 109 lines on the common layer and 109 lines on the top layer are needed for implementing TCP-based communication on the Linux platform. 568 lines of codes which are noted after the line numbers of TCP and Bluetooth on the common layer are shared among socket-based communication. For this reason, if a developer adds a socket-based communication method, 568 lines of codes can be shared for its implementation. Serial communication support is totally different to socket-based communication, so 478 lines of codes in total are written for implementation. Unfortunately, the minimized communication code synthesis technique described in section 3.5 is not applied in a constrained device target, so the core layer codes for supporting a communication method are quite big compared to that of an unconstrained device target.

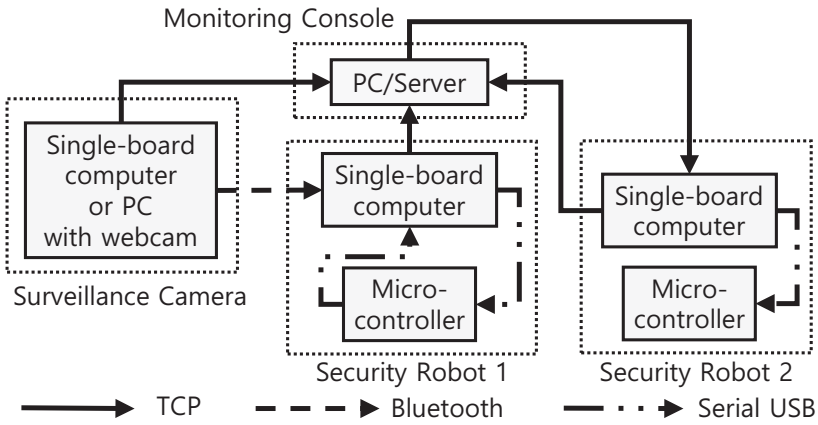


Figure 3.12: Top-level specification and hardware association of the surveillance system

In summary, at most several hundreds of lines are needed for applying a new platform or network on our program synthesis tool. Therefore, the required codes for extending a platform or network are very small compared to the total lines of codes in Table 3.2.

3.6.2 Program Synthesis for the Surveillance System Example

The proposed software development method is used to synthesize the parallel and distributed program for the surveillance system which is introduced as a motivational example in section 3.1. Figure 3.12 shows the model-based task level specification at the top level, based on Fig. 3.1. It consists of 6 super tasks that are mapped to 6 processing elements. A PC or a server is used as the monitoring console, and a laptop or a single-board computer with a webcam plays the role of a surveillance camera. In this example, we use two robots that have two processing elements each: a single-board computer and a micro-controller. Note that three types of communication methods are used in this example as shown with different arc styles in the figure. TCP communication is used between the monitoring console and the other devices through Wi-Fi. The surveillance camera is connected to security robot 1 via Bluetooth to send an object detection result. The connection between two processing elements, a single-board computer and a micro-controller, in each robot is made through a serial USB line.

Figure 3.13 represents the hierarchical task-level specification of the surveillance system. In the surveillance camera that is composed of three tasks, the *ObjectDetection* task detects a person from the camera and sends the detection result to security robot 1. Also, it transmits the captured image to the monitoring console. The monitoring console receives camera images from two security robots and the surveillance camera and encodes and stores them in the *x264Encoding* task. The compute-intensive *x264Encoding* task has a child task graph inside to explicitly express the parallelism of the algorithm. The *Command* task is defined independently with the capability of sending commands to security robot 2 for remote control. Security robot 1 receives the detection result from the surveillance camera and also receives sensor data from its micro-controller. After gathering the information, it determines an action to do. The behavior of security robot 1 is described by a control task, *ControlTask*, whose behavior is represented by a finite state machine. The execution status of the *WheelControl* and *LEDControl* is managed by *ControlTask* and sends the motor and LED control commands to the micro-controller. Besides the basic motor control functions of security robot 1, security robot 2 is equipped with extra functions such as remote control, object tracking, and random driving. For object tracking, the Tracking-Learning-Detection (TLD) [28] algorithm is specified inside the *Tracking* task and used to follow security robot 1. If the remote-control command is received from the monitoring console, security robot 2 changes the state of its *ControlTask* to the *Remote* state.

In this example, a task mapping decision could be easily made. With the given task codes that are manually made and verified a priori, program synthesis is performed for 6 processing elements. For the surveillance camera, we used a laptop with a webcam to take an image, and a Coral USB accelerator [29] is used for detecting objects. We used a Linux-powered PC as a monitoring console, and two TurtleBot3 [30] robots are used to act as security robots. Each robot consists of Raspberry Pi 3 Model B+ (ARM Cortex-A53, quad-core, 1.4 GHz) as a single-board computer and OpenCR 1.0 [31] board (ARM

Table 3.5: LOC of synthesized program

Processing elements	Total LOC	Task code (%)	From template code (%)	From library code (%)
Security Robot 1 (Raspberry Pi)	20,459	2.698%	9.023%	88.279%
Security Robot 1 (OpenCR 1.0)	4,868	3.492%	11.319%	85.189%
Security Robot 2 (Raspberry Pi)	23,863	10.837%	14.315%	74.848%
Security Robot 2 (OpenCR 1.0)	4,510	1.840%	6.208%	91.951%
Surveillance Camera	19,036	1.056%	6.577%	92.367%
Monitoring Console	29,022	33.096%	7.008%	59.896%

Cortex-m7, 216MHz) as a micro-controller. These robots have a laser distance sensor (LDS) and a raspberry pi camera which can be connected to a Raspberry pi board, and two motors are controlled by OpenCR 1.0 board. Also, security robot 1 has additional sensors such as a motion detector and a distance sensor. We built and ran this system, and confirmed that the surveillance system is normally operated. Figure 3.14 shows the monitoring console screen during the demonstration of the experiment.

Table 3.5 shows some statistics on the synthesized program. It shows that the library code takes the largest portion in the synthesized program from 60% to 92%. The percentage of the task code that the programmer should write manually takes a little portion of the entire program. If we can use the line of code (LoC) as an indirect metric to measure the improvement of software design productivity, the proposed method reduces the software development efforts significantly, besides the reduction of the verification effort by compile-time static analysis.

3.6.3 Remote GPU-accelerated Deep Learning Example

Another example is a remote deep learning example. Because our program synthesis tool is targeted on not only a homogeneous multiprocessor system but also a heteroge-

neous multiprocessor system, we implemented a CNN ResNet [32] Inference with 152 layers using GPU. Besides, we used two heterogeneous devices to describe a distributed system. One device represents a low-performance edge device that has limited hardware resources. The other device represents a server with a high-performance GPU. The overall software system is depicted in Fig. 3.15 (a). Note that this task graph uses an SDF/L to describe multiple iterations of the same convolutions to show a compact view of 152 layers, and the internal view of each residual block is shown in Fig. 3.15 (b). All tasks located in the high-performance device in Fig. 3.15 (a) are mapped onto GPU, and tasks in the low-performance device are mapped onto CPU. We experimented with this example on an NVIDIA Jetson AGX Xavier board [33] as a low-performance device and a server with NVIDIA RTX 2070 GPU. To compare the benefit of using the remote high-performance device, we first executed the example on the Xavier board, so all tasks are mapped onto the processing components in the Xavier board. Because the Xavier board also provides a GPU with 256-core as a co-processor, we can accelerate the deep learning example in the board. The total execution time of classifying 1,000 images on the single board is 1 minute and 6 seconds on average. However, using remote GPU only takes 20 seconds on average, which means using a remote high-performance device is three times faster than running on the single board. Therefore, the code generation framework enables a distributed system design for software development, so it can provide options to choose single-device or multi-device implementation without additional development costs such as CPU-to-GPU memory copy logic and network programming.

3.7 Document Generation

The code generation framework generates not only codes but also documents. It generates a header file with Doxygen [34] comments, so a user can run Doxygen to get full documents in HTML or Latex. First of all, it generates basic documents based on specifications of models, hardware information, and mapping information. If a user writes a detailed explanation in HOPES, the code generation framework inserts a detail into the documents. Fig. 3.16 (a) shows a basic document with an application task graph. Because the code generation tool uses a DOT provided by [35], it generates a hierarchical diagram of specified applications. Fig. 3.16 (b) shows individual task information, and it contains the task and its port information. In addition, the framework also includes Doxygen comments from multiple layers of codes written in the library code, so an application developer helps to understand the detail of generated codes as long as he or she wants to know the codes.

3.8 Related Works

There exist commercial tools that generate embedded software code from model-based application specifications. MathWorks Simulink Coder generates a C program from task-level specifications based on the Simulink model and Stateflow. While dSPACE's TargetLink also generates code through Simulink models, it is tailored to ECU software code generation. Since the Simulink model is basically a simulation model, functional correctness of the specification should be verified by simulation without any static analysis. Another commercial tool is the National Instruments' LabVIEW C Generator that generates a C code from a task graph specification. While these tools support a multi-core processing element, they do not support a distributed embedded system where multiple devices cooperate to perform an application. Also, no fault-tolerance method is provided.

Automatic code generation from formal models has been extensively studied in academia. The research team of Ptolemy II has proposed several code generation methods based on the data flow model ([36, 37, 38]) and on the combination of the FSM model and a dataflow model [5]. To utilize the static analysis capability of the SDF model, some researchers proposed to convert the Simulink model to SDF and generate code from the converted SDF model [39]. Code generation from dynamic dataflow models is also researched [40]. Previous researches on code generation from dataflow models are mostly focused on the optimization of the synthesized program in terms of memory requirements and real-time performance on a single processing element. The most closely related to our method is the work [3] that proposes the task-level specification adopted in this work. While it proposes a code generation flow from task-level specification, only preliminary experiments are conducted to prove the idea. To the best of our knowledge, there is no previous work that supports distributed embedded systems with diverse communication methods and fault-tolerant code generation.

There exist researches on the program synthesis from a UML-based model. While

UML does not have formal execution semantics, fUML (Foundational UML) [41] is standardized to support execution of UML. To describe fUML in textual notation, Alf (Action language for fUML) [42] is used. [43] and [44] generate C++ codes from Alf and mainly focus on translating Alf expression to C++ expression. Another work is [45], and it modifies the front-end of GCC to directly translate fUML models to target binary codes. Also, its generated assembly code size is compared with that from the C++ programming language. The works related to fUML mainly focus on expressing the structural behavior of applications, not parallel and distributed application behaviors.

There are also automatic synthesis methods based on UML profiles. Some researchers use SysML [46] to design and synthesize codes for embedded systems. Authors of [47] extend a basic Y-chart approach to ψ-chart approach which includes a communication model targeted on parallel and distributed embedded systems. However, their target applications are focused on signal processing platforms, and they consider only an intra-device communication code synthesis. [48] uses a SysML activity diagram to generate codes running on an ARM Cortex-M processor. Although it supports multi-threaded code generation running on a real-time operating system, their target platform only uses a single processor, not a multi-core embedded system. [49] synthesizes a functional behavior that is generated from Simulink Coder with platform-dependent communication and synchronization codes from Acceleo [50] template codes. Though the work considers real-time embedded systems with multiple ECU components, its supported RTOS, OSEK, is a single processor system running on each ECU component. Also, it does not support a network-required distributed system where multiple devices cooperate to perform an application.

MARTE [51], a UML profile for real-time embedded systems, is applied in various researches. [52] uses UML and Alf for structural and functional behaviors, and MARTE is used to describe a schedulable task with mapping and timing information. Also, the work is targeted on multi-core embedded systems. However, it cannot express distributed

embedded systems with networks. Another MARTE-based research is [53] which is focused on communication and concurrency of automatic code generation. It models communication media with MARTE and defines a channel to be semantic of communication and concurrency. Although it shows examples with stream-based parallel applications, describing control behaviors is not considered.

Another UML-based model is ThingML ([54, 55]), a modified version of UML used for specifying an IoT Application. It generates software codes for distributed heterogeneous systems. The authors of [56] proposed another technique based on the ThingML model, presenting more detailed specifications related to communication. The target system of ThingML is a distributed embedded system, similarly to our framework. However, this model is suitable to describe a control-oriented application but not adequate to express the parallelism of a compute-intensive task such as object detection and tracking task. Also, fault-tolerant code synthesis is not considered. Another code generation method from a UML-based model can be found in [57] which generates codes from the UML Statechart, focusing on the control-oriented application.

Authors from [58, 59] introduce a domain-specific language to specify safety rules for robot applications, and safety monitoring codes are generated from its specified rules. They apply this technique on ROS and some robots with micro-controllers [60]. However, the scope of this research is limited to adopting safety monitoring. [61] is based on ScicosLab [62] model which is used for modeling scientific software. It provides a toolset to generate not only functional and control codes based on Scicoslab but also GUI and communication codes. While they support multiple targets such as Linux, Windows, and OSEK, their system design is restricted into a single device with a multi-core system.

Some communication APIs are defined to make it easy to write a distributed program: MPI [63] and UCX [64] are two examples. Since they primarily focus on high performance distributed systems, they are not suitable to use for low-power/low-performance networks. ThingML [54] which is mentioned earlier supports plugins for

low-performance communication networks such as Bluetooth, UART, or IoT-related protocols. Another approach is to generate middleware that manages communications between heterogeneous devices [8]. There are some proposals to use a new modeling language to generate communications codes for distributed systems ([10, 11]). However, these researches are focused on a manual description of external communication only.

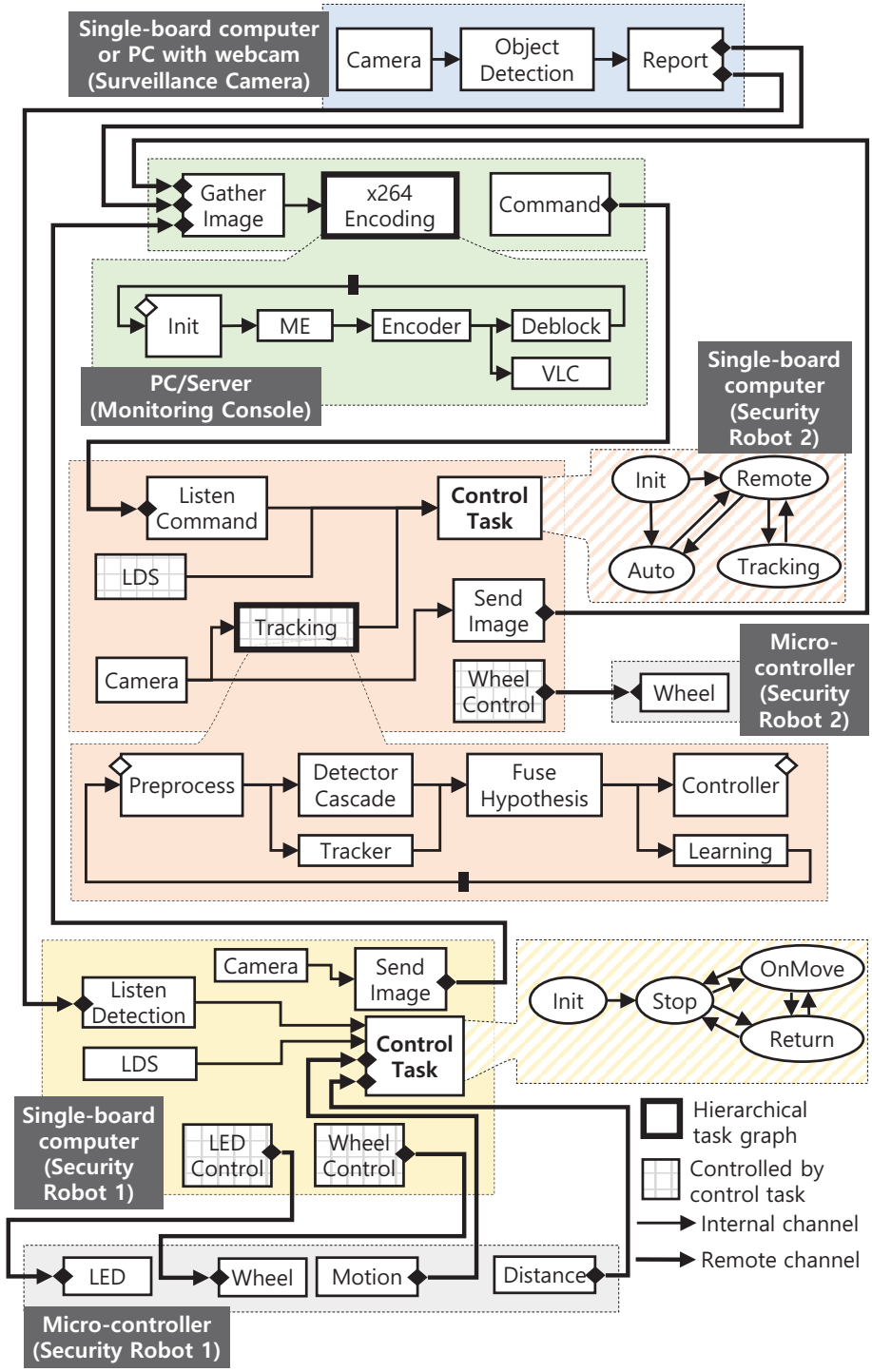


Figure 3.13: Entire task graph of surveillance system

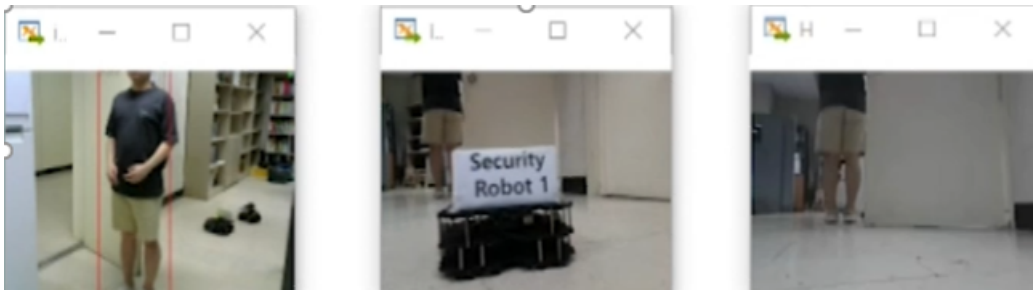
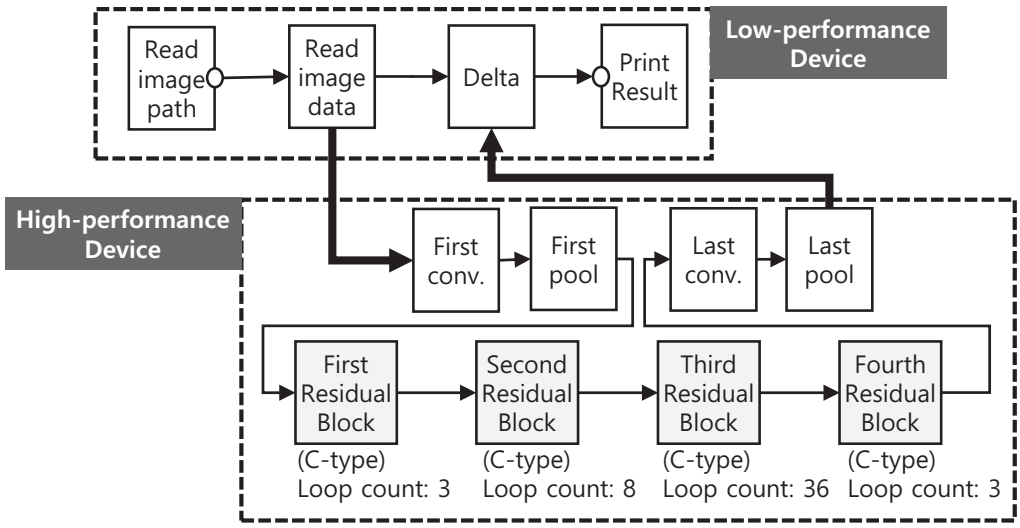
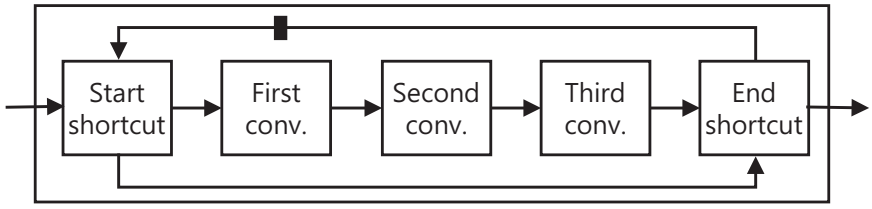


Figure 3.14: Monitoring console screen during the experiment

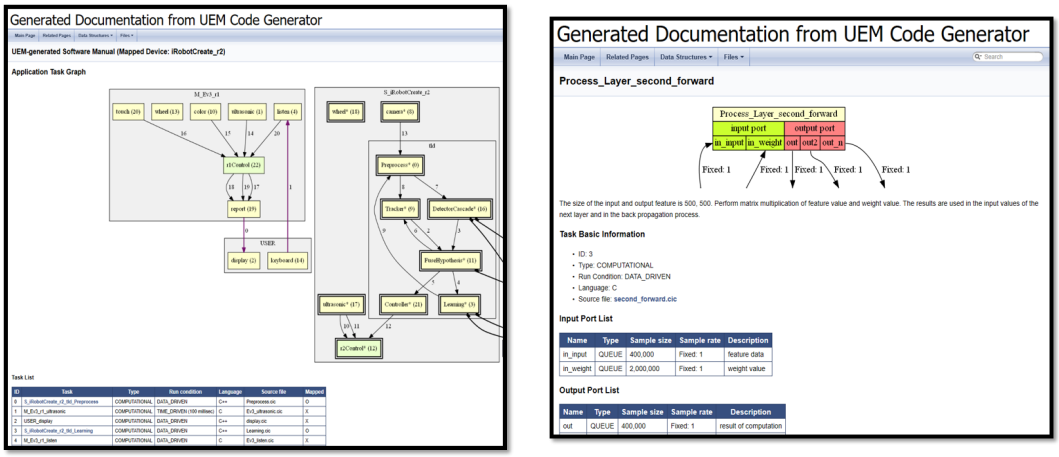


(a) Task graph of ResNet inference example with two heterogeneous devices



(b) Internal task graph of each residual block in (a)

Figure 3.15: Task graph specification of ResNet inference with 152 layers



(a) Full application task graphs **(b) Single task**

Figure 3.16: Generated documents from the code generation framework

Chapter 4

Model Transformation for Fault-tolerant Code Synthesis

4.1 Fault-tolerant Code Synthesis Techniques

Two popular methods to tolerate transient faults are re-execution and replication. The former is to execute the same code multiple times until no error is detected in the validation code that is appended at the end of the code. The latter is to execute multiple copies of the same code concurrently on different processors and choose the final results via majority voting that is performed after collecting the results from all replicas. The number of replicas is usually set to an odd number to avoid a tie in voting. Suppose we want to make task B tolerant of transient faults in the simple task graph shown in Fig. 4.1. In the figure, x and y mean the number of data samples consumed and produced by task B at each invocation.

In the proposed methodology, we transform the task graph using a loop task and synthesizing the program from the transformed SDF/L graph. Since such graph transformation and program synthesis is done automatically, we can easily change which fault-tolerant method to be applied to which tasks. Also, the application programmer is relieved of the non-negligible burden of ensuring fault tolerance in the early phase of software development, task-graph specification and task code definition.

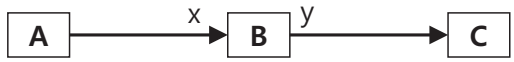
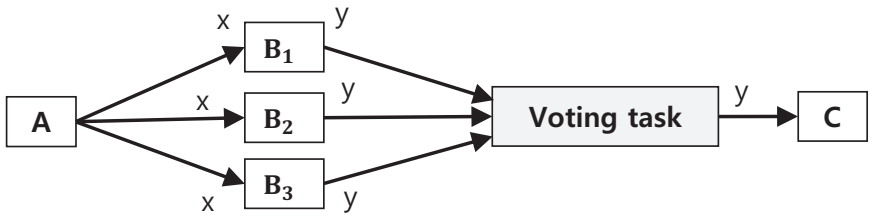
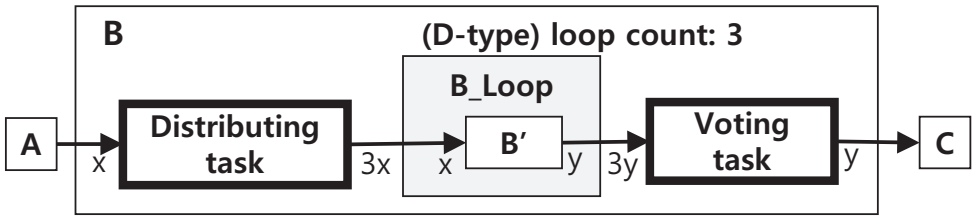


Figure 4.1: A task graph example to apply a fault tolerance technique



(a) The behavior of active replication applied to task B

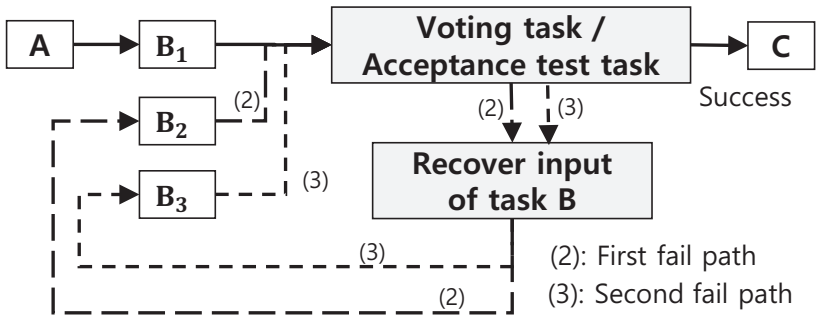


(b) The translated SDF/L graph for active replication

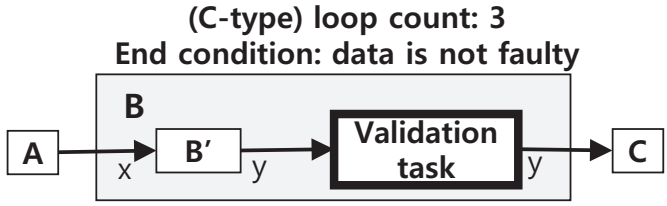
Figure 4.2: Fault-tolerant scheme based on active replication

The behavior of active replication is displayed in Fig. 4.2 (a) where task B is instantiated three times and all instances of task B receive the same input data and send the results to the voting task. Such behavior can be specified with a D-type loop in the SDF/L model as shown in Fig. 4.2 (b). Task B is wrapped in a D-type Loop task called B_Loop, and the loop count of B_Loop is set to 3 because the results of three task instances will be compared. Two simple tasks are added before and after task B_Loop. One is to copy the input data sample three times and distribute them to three task B instances, and the other is to perform majority voting among the received three results.

The behavior of the re-execution method is expressed in Fig. 4.3 (a). After executing the first instance of task B, the correctness of the result is checked in a certain way. If it is correct, the result is passed to task C. Otherwise, the program follows route (2) to re-execute another task instance. If B_2 fails to pass the acceptance test, it follows route (3) which is similar to route (2) except executing task instance B_3. In case there is no easy way of testing the output correctness, we may use by re-executing the task more



(a) The behavior of re-execution applied to task B

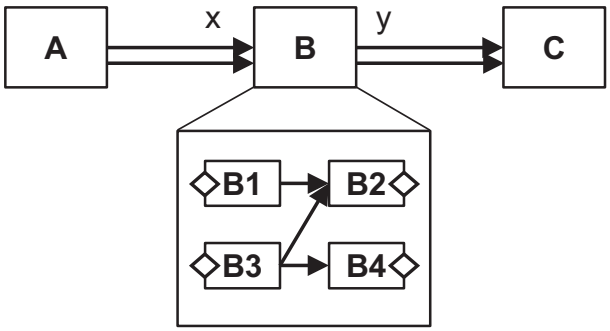


(b) The translated SDF/L graph for re-execution

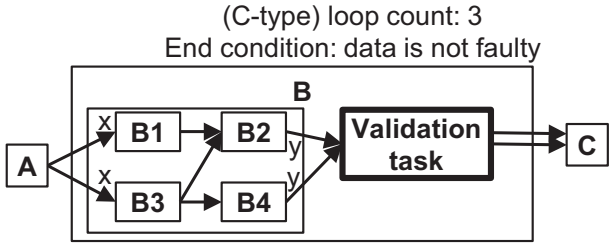
Figure 4.3: Fault-tolerant scheme based on re-execution

than once. If two out of three results are identical, the result is considered correct. Such a scenario of re-execution can be expressed by a C-type loop as shown in Fig.4.3 (b). Inside the C-type loop, one task is added automatically to check the correctness of the result by designating the added task to set the exit-flag, and the added task may perform majority voting after the C-type loop is iterated as many times as the loop count parameter at most, which is 3 in this example.

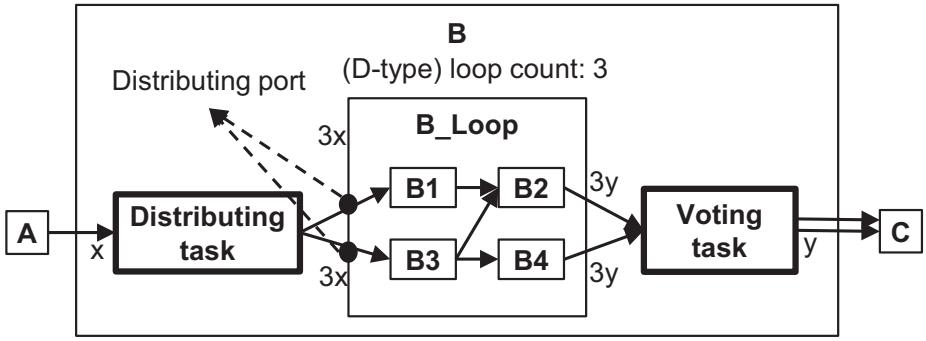
We can configure fault tolerance parameters such as the number of task instances, or the type of the validation task. By modifying the number, the degree of task fault tolerance can be adjusted. For example, if a user sets a re-execution count 3 to 5, a task can tolerate two errors among five executions. Then, the application execution time will increase. Basically, thick-bordered tasks in Fig. 4.2 (b) and Fig. 4.3 (b) such as a distributor task and a validation/voting task are automatically generated by the code generation framework. These validation tasks use a majority voting to select the output. If an application developer has own fault checking mechanism, he or she can replace a validation task to a user-defined task.



(a) A task graph example to apply a fault tolerance technique to a task graph



(b) The translated SDF/L graph for re-execution



(c) The translated SDF/L graph for active replication

Figure 4.4: SDF/L conversion on the task graph to apply fault tolerance

This fault-tolerance technique is applied to not only a single task but also a task graph. Fig. 4.4 shows the example of applying each fault tolerance method on a task graph. Although multiple input ports and output ports are connected to the outside of the task graph B in Fig 4.4 (a), UEM supports multiple ports and channels between two tasks so that the task graph B can be simplified as a task B with two input ports and output ports in the upper task graph. Applying a fault tolerance technique to a task graph uses only a single validation task for re-execution and two extra tasks for active replication, so the extra overhead is smaller than applying fault tolerance techniques to all the individual tasks one by one. However, the fault tolerance capability is also reduced because a task-graph-level fault tolerance technique cannot tolerate multiple errors occurred from different tasks in the task graph on different replications.

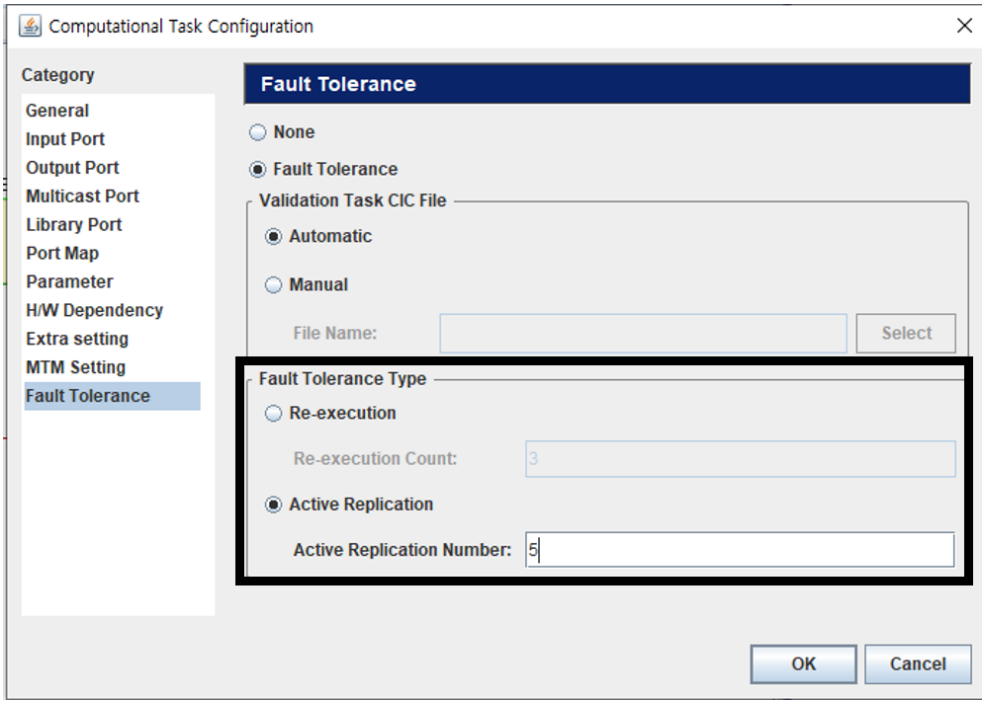


Figure 4.5: A dialog in HOPES to apply a fault tolerance technique

4.2 Applying Fault Tolerance Techniques in HOPES

The technique can be applied by changing the original dataflow model to the SDF with a loop structure model, so the fault-tolerant code synthesis is not applied in the program synthesis step in Fig. 2.1. It can be applied right after the application specification step. Because the model transformation is performed at the early stage of the development process, static analysis and performance estimation can be applied to the fault-tolerant dataflow models. As shown in Fig. 4.5, fault tolerance can be applied by clicking some buttons and changing some parameters, so application developers can easily use the technique without background knowledge.Deciding on which task to apply fault tolerance techniques is introduced in [65], and the paper chooses proper tasks and techniques from mixed-criticality tasks with real-time constraints.

Table 4.1: Comparison of applying fault tolerance with manual implementation

Fault tolerance method	Efforts to apply
Proposed Approach	3 clicks, 1 word typing
Manual active replication (single task)	154 lines
Manual re-execution (single task)	56 lines
Manual active replication (task graph)	154 + 219 lines
Manual re-execution (task graph)	56 + 219 lines

4.3 Experiments

4.3.1 Development Cost of Applying Fault Tolerance

Using HOPES UI, applying fault tolerance to tasks is easy with a few clicks. To prove the benefit of development productivity, we implemented active replication and re-execution manually and calculated the number of lines. Table 4.1 is the result of manual implementation. Note that active replication takes more efforts compared to re-execution because active replication needs task creation and processor mapping logic which is tightly dependent on OS platforms. Moreover, if an application developer wants to apply fault-tolerance on multiple tasks, he or she needs to implement task communication codes between tasks. Manual implementation for applying fault tolerance requires up to 400 lines of codes, and an application developer expects to know background information about multi-task and communication programming which is hidden by UEM.

4.3.2 Fault Tolerance Experiments

To verify the improvement of fault tolerance of a program against transient faults by the proposed method, a fault injection experiment is conducted. We developed a fault injection tool [66] which is another contribution to this work. This fault injection framework can inject a fault to both the kernel and application layer of the Linux system. The detailed description is shown in Chapter 5. The test example used for fault injection is the *x264Encoding* task shown in Fig. 3.13 of the monitoring console. It has a child task graph with five tasks, and fault injection is applied to the *ME* task at a specific location.

(a) Original image **(b) Fault-injected image**

Figure 4.6: Original image and fault-injected image

Table 4.2: The number of injected faults with varying fault tolerance configurations

Configuration \ Injection rate	2%	5%	10%
No fault tolerance	601	1,421	2,952
Active replication (3 cores)	1,849	4,459	8,821
Active replication (5 cores)	2,941	7,455	14,690
Re-execution (3 times)	1,229	3,151	6,483
Re-execution (5 times)	1,849	4,730	9,841

The motion estimation (ME) task takes more than 60% of the encoding time. Injected faults cause spots in the output image so that the relative degree of error occurrence can be noticed in the output image displayed by the *Deblock* task, which is illustrated in Fig. 4.6.

Table 4.2 shows the number of faults injected, and Fig. 4.7 shows the number of error occurrences as we vary the fault injection rate and fault tolerance configuration. With the same fault injection rate, more faults are injected when the fault tolerance method is applied. Nonetheless, it could be observed that the proposed fault tolerance method reduces the error occurrence rate from 70% to 99.5%. As we increase the number replications on the processing cores or the number of re-executions, the error occurrence rate is decreased noticeably.

Another set of experiments is conducted to find how long the execution time is increased by applying the fault tolerance methods on the Intel i9-9900K (3.6 GHz) machine

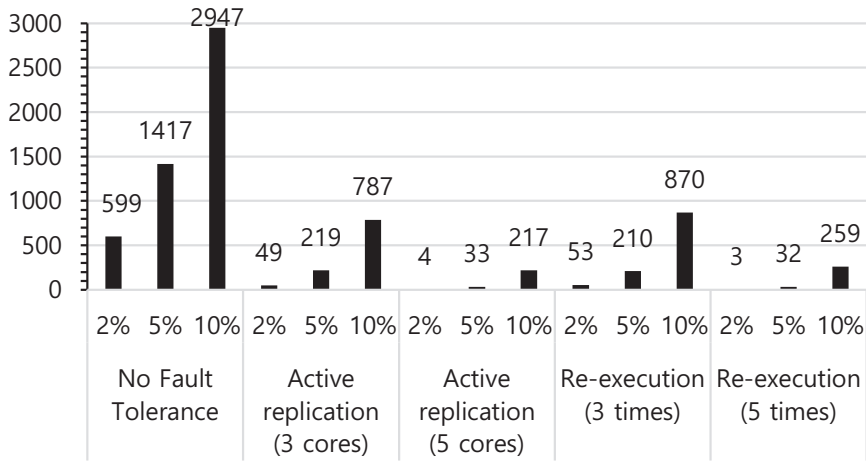


Figure 4.7: The number of error occurrences with varying fault tolerance configurations

Table 4.3: Fault tolerance settings and their labels

Label	Fault tolerance settings
A	No fault tolerance
B	ME (Active replication, 3 cores)
C	ME (Active replication, 5 cores)
D	ME (Re-execution, 3 times)
E	ME (Re-execution, 5 times)
F	ME (Active replication, 3 cores), Encoder (Active replication, 3 cores)
G	ME (Re-execution, 3 times), Encoder (Active replication, 3 cores)
H	ME + Encoder (Clustered, active replication, 3 cores)
I	ME + Encoder (Clustered, active replication, 5 cores)

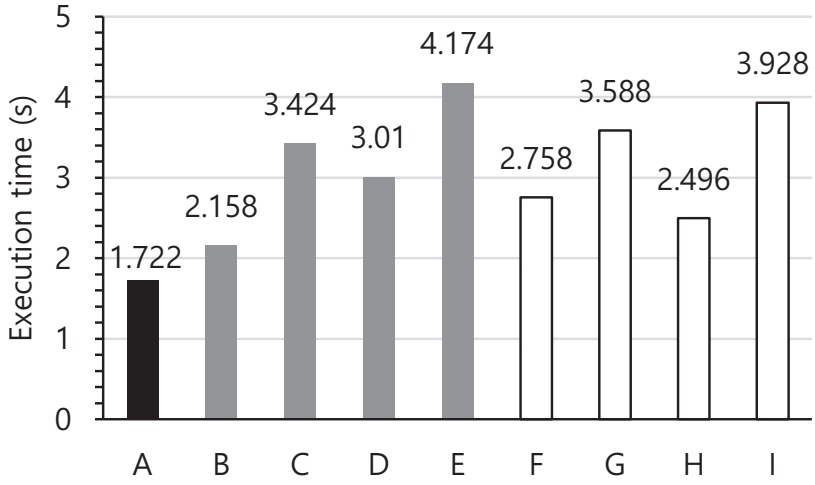


Figure 4.8: Comparison of total execution time by fault tolerance settings

with 64GB memory and Ubuntu 18.04 operating system. Nine fault tolerance settings are defined, as shown in Table 4.3. Fault tolerance techniques are applied to task ME only or both tasks ME and Encoder together. Fault-tolerant code generation can be applied to a cluster of tasks as well as an individual task. In settings from F to J, a fault-tolerance method is applied to the cluster of the ME and the Encoder task. Note that a re-execution method is not applied to task Encoder because the task has an internal state.

Figure 4.8 shows the total execution time with various fault-tolerance settings. The re-execution method increases the execution time more than the active replication method with the same degree of fault tolerance, as expected. Comparison with setting F and H reveals that the execution time of cluster tasks is shorter than the sum of the individual tasks with the same fault tolerance method.

4.4 Random Fault Injection Experiments

Because the previous experiment's fault injection is targeted on a specific location of each task, we performed a random fault injection experiments on fault-tolerant memory spaces. To conduct this experiment, we used a ptrace-based fault injection tool which

Table 4.4: Error occurrences of random fault injection (3,000 runs)

	No fault tolerance	**Active replication (3 copy)**
Number of faults injected	599,684	873,947
Normal	1,846 (61.53%)	2,296 (76.53%)
Hang	42 (1.40%)	80 (2.67%)
Crash	355 (11.83%)	422 (14.07%)
Silent data corruption (SDC)	757 (25.23%)	202 (6.73%)

is also developed by us. As the ptrace-based fault injection method is already studied in [67] and [68], this is a representative run-time fault injection method for application-level programs. The benefit of using a ptrace-based fault injection tool compared to a GDB-based fault injection tool is a low latency when injecting faults, so many faults can be injected with negligible delay on a victim program. The target program manages most of its data on the BSS section, so we injected bit-flip faults on the specific range of BSS section area which is related to fault tolerance such as fault-tolerant tasks and generated tasks.

Table 4.4 shows the number of errors from 3,000 runs. We used the same x264 encoding example which is delivered on the previous section and applied a 3-copy active replication on the task graph which consists of two tasks: ME and Encoder. As a fault tolerance is applied on a task graph, there is only one distributor task and majority voting task. The errors are categorized into 3 types: hang, crash, silent data corruption (SDC). We detected a hang if the program is not ended twice longer than the expected execution time, and a crash is detected when the program is exited unexpectedly with a signal such as segmentation fault. Because this application generates an output x264 file, we checked the output file with the expected output file. If two files are different, we consider this case as an SDC. According to the result, errors from SDC are drastically reduced compared to the program with no fault tolerance. However, hang and crash are increased. The reason why crash errors more frequently happen is that an index value used for accessing an array is bit-flipped. The fault-tolerant program may contain three times more related variables, so the crash rate is increased. Hang occurs since data transfer is not

happened because of errors on channel IDs. Also, the fault-tolerant program has more channel ID variables, and this reason causes more hangs. To protect the program from hang/crash errors, multi-process code generation is needed since an error affects an entire multi-threaded process. Multi-process code generation can be future work to enhance the fault tolerance from critical errors.

4.5 Related Works

There exist software-based fault tolerance techniques that duplicate instructions to enhance resilience from error. SWIFT [13] duplicates arithmetic and logical operations with unused resources. SWIFT-R [14] is an enhanced version of [13], and it triplicates instructions and performs a majority voting to recover data. nZDC [69] duplicates a control flow and load instructions and rechecks the store data to tolerate errors that are made by branch and load/store operations. ELZAR [70] uses a SIMD (Single Instruction Multiple Data) instruction to operate redundant behavior to reduce the execution overhead. These instruction-level fault tolerance techniques require a modified compiler which means target-dependent to the CPU architecture, and fault tolerance is applied to the whole application.

The second category of software-based fault-tolerance techniques is duplicating the main thread of an application. [15] introduces SRMT (Software-based Redundant Multi-Threading) which consists of a leading thread and a trailing thread. A trailing thread is a duplicated version of a leading thread, and it receives data from a leading thread and checks the data. If a trailing thread finds an error, it throws an ack to a leading thread. [16] is a modified version of SRMT. A leading thread and a trailing thread both load data separately, but a leading thread only stores data. A trailing thread checks data stored by a leading thread to detect errors. Because of this operation, it can detect unwanted write errors. [17] provides an error recovery method with one main thread and two redundant threads. Depending on the fault location of a thread, it restores a state. If the main thread causes a fault, it restores a state and data and retries store operation. These thread duplication-based fault tolerance methods also require a modified compiler. Also, fault tolerance is applied to the whole application.

There is a framework that provides fault tolerance for developing applications. [71] is a fault-tolerance framework targeted on the BOSS real-time operating system. It is based on C++ and provides classes to inherit and apply a fault tolerance per each thread.

It supports multiple fault tolerance methods and other necessary features such as a voting thread or fault tolerance thread scheduler. The scope of applying fault tolerance is similar to our proposed approach, and the framework makes it easy to apply various fault tolerance methods. However, it is worked as a thread, not a formal model, so static analysis and performance estimation cannot be used for this framework.

A model-based application-level technique is proposed in [72]. Similarly to our proposed technique, it covers the entire flow from initial model-based specification, formal analysis, and program synthesis, targeting a distributed embedded system. It also can generate fault-tolerant codes based on Triple Modular Redundancy (TMR); active replication, passive replication, and semi-active replication can be selectively applied in the program synthesis phase. However, it uses a meta-model for initial specification and performs model-to-model transformation in the subsequent steps while the proposed method starts with the dataflow based task graph. It mainly focuses on control-oriented applications while ours consider both control tasks and compute-intensive tasks. Also, fault-tolerant method in [72] is applied to the entire program, but our framework can selectively apply fault tolerance techniques to some tasks with different configurations.

Chapter 5

Fault Injection Framework for Linux-based Embedded Systems

5.1 Background

5.1.1 Fault Injection Techniques

Fault injection can be performed by injecting faults directly to hardware, or by emulating faults in software. Direct fault injection to hardware has been made by pin-level injection [73, 74] which injects faults through direct physical contact to the target chip, ion radiation [75], electromagnetic [76], or laser [77]. This approach, however, requires us to modify the target device or add an extra hardware module, which is not always possible. To overcome this difficulty, software-implemented fault injection (SWIFI) techniques have been proposed to emulate hardware faults by software [78, 79, 80].

SWIFI techniques that perform fault injection in the code-level of programs can be used not only for emulating hardware faults but also for robustness testing of software [81]. SWIFI for hardware fault emulation is based on the examination of how hardware faults affect the software. On the other hand, SWIFI for robustness testing, also known as Software Fault Injection (SFI), is focused on test coverage increase [82] and detection of software bugs [83, 84]. According to [85], emulated faults are classified into three types—data errors, interface errors, and code changes. Data errors are related to hardware

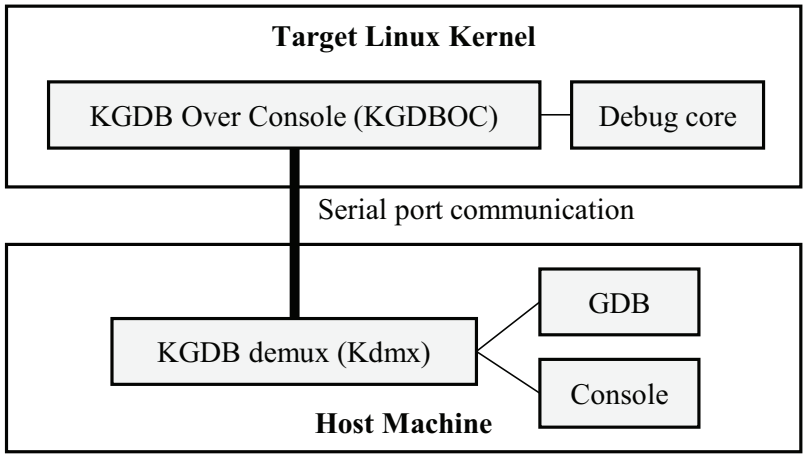


Figure 5.1: The overview of a kernel debugging environment with KGDB

faults such as bit-flips or network data corruption. Depending on which kinds of interface faults affect a component, interface errors can be induced by a fault in the hardware, software, or environment. Code changes are software faults that emulate software bugs based on the field data or mutating operators. Our tool emulates hardware faults to raise data and interface errors.

5.1.2 Kernel GNU Debugger

Kernel GNU Debugger (KGDB) is a kernel debugging tool which interacts with a remote GDB. KGDB is officially supported in the Linux kernel since version 2.6.26. To use KGDB, a separate host machine is needed that communicates with the target system through serial port communication. The kernel debugging environment with KGDB is shown in Fig. 5.1.

In the target Linux kernel, KGDB consists of two modules. One is ”debug core” and the other KGDBOC (KGDB Over Console). The former is responsible for kernel debugging while the latter is used to interact with a remote host machine. The KGDBOC module communicates with the KGDB demux (Kdmx) module in the host machine by a serial port communication. The Kdmx module creates two pseudo-terminal ports so that

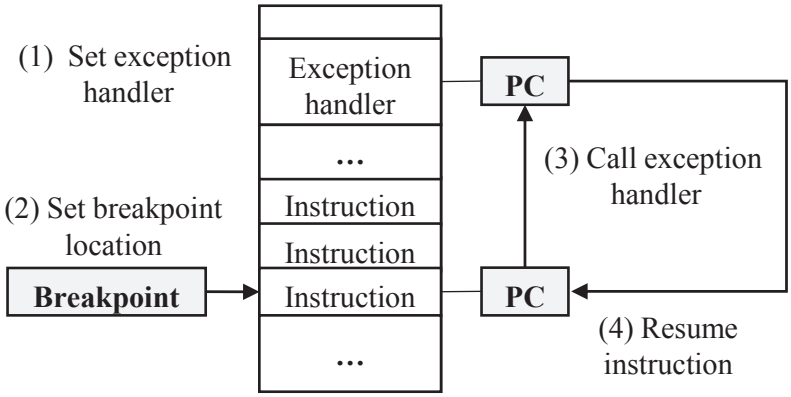


Figure 5.2: The conceptual diagram of hardware breakpoint mechanism

```
1 // Set breakpoint address
2 asm volatile("mcr p14, 0, %0, c0, c0, 4" : : "r" (addr));
3 // Enable breakpoint
4 asm volatile("mcr p14, 0, %0, c0, c0, 5" : : "r" (0x21e7));
```

Figure 5.3: Code example to set and enable a hardware breakpoint

the connection can be made with the GDB program and the console program such as minicom [86] via a single serial port.

5.1.3 ARM Hardware Breakpoint

ARM supports hardware breakpoints that are used for suspending the execution of the system on a specific instruction or memory access. For hardware breakpoints, we have to register instructions, called breakpoints, or memory locations, called watchpoints, as illustrated in Fig. 5.2. When the system meets a registered instruction, a prefetch-abort exception occurs. On the other hand, a data-abort exception occurs when a registered memory location is accessed. In a multiprocessor system, breakpoints and watchpoints need to be set on each CPU individually.

As the hardware breakpoint is a platform-dependent feature, assembly instructions are used to register and enable breakpoints or watchpoints. Figure 5.3 shows a code example to set and enable a hardware breakpoint for an ARM processor. The last argument of *mcr* instruction specifies the special purpose debug register: 4 to set and 5 to enable

the breakpoint. To set a breakpoint, we provide the address of the breakpoint while we specify the adequate flags (0x21e7) that the ARM processor defines to enable the breakpoint.

5.2 Fault Injection Framework

5.2.1 Overview

For kernel-level fault injection, the proposed fault injection framework consists of two complementary fault injection methods based on KGDB and hardware breakpoints. Fault injection through KGDB has a very desirable feature to manipulate data without changing any kernel code. If the kernel is built with the KGDB option on, the GDB can refer to the debug information augmented to the kernel image. It can insert a breakpoint at any line in the source code and access any data structure of the source code as well as memory locations. For instance, access to complex data such as a multi-dimensional pointer to a member variable of structure can be made easily. Also, this method has the potential to support other architectures because the Linux kernel supports KGDB not only on ARM but also on other processors such as x86_64, SPARC, PowerPC, and MIPS. While KGDB provides convenient means of fault injection without kernel modification, the execution time overhead is high because of serial communication delay. Also, it cannot emulate a timing-related error.

Our framework provides two ways of using hardware breakpoints for fault injection, generic fault injection and custom fault injection. For generic fault injection, it adds a separate fault injection module to the kernel that has configurable options to test a range of different cases. We may change the fault locations and fault types at run-time without recompilation once the kernel is built with the fault injection module. The fault types it supports are bit-flip error and timing delay. Since it incurs negligible execution overhead, breakpoints can be set in a time-critical code section such as an interrupt handling routine, which should be avoided in the KGDB method.

The custom fault injection method is similar to the compile-time fault injection technique in that fault injection position is predefined by the inserted code. Similarly to the generic fault injection method, it adds a specialized fault injection module to the ker-

nel, but with customized configuration options. It gives most freedom to the tester to support other fault types that the generic fault injection method and the KGDB method cannot support. For instance, we can customize the device interface software to emulate the failure of a specific hardware device. In this work, we use the custom fault injection method to emulate the failure of an embedded multi-media card. Even though it has low execution overhead as the generic hardware breakpoint method, it requires kernel recompilation whenever a new fault location and a new fault model is defined. Hence this method is optionally included in FIFA only when we need to design a complicated data manipulation logic that the other methods cannot provide.

For injecting a fault to an application layer, a GDB and a GDB server are used to inject faults on an application located at a local or remote device. This method also has same merits and demerits of the KGDB method. a GDB prolongs the latency of the application noticeably. However, it enables us to inject faults regardless of the system architecture running the application and faults on multiple applications running on remote devices simultaneously. Also, we can inject a fault at a specific location with debugging symbols such as source code file, line number, and variable name. This feature is useful for debugging by repeating the same fault scenario.

Fig. 5.4 illustrates the architecture of the fault injection framework. It includes four software components: fault injection manager, kernel fault injector, hardware breakpoint fault injection module, GDB-based application fault injector.

5.2.2 Architecture

Figure 5.4 illustrates the architecture of our fault injection framework. In addition to the KGDB environment, as shown in Fig. 5.1, it includes four additional software components: fault injection manager, kernel fault injector, hardware breakpoint controller, hardware breakpoint fault injection module, and GDB-based application fault injector.

The fault injection manager manages input/output files and coordinates multiple

experiments running on different devices. It communicates with the kernel fault injector to send configuration data and receive results. The fault injection manager is designed as a target-independent module, meaning that it can communicate with various kinds of fault injectors such as application-level fault injectors or non-Linux system fault injectors. Furthermore, it can perform multiple fault injection experiments to enable testing on multiple devices in parallel. For each experiment, the fault injection manager creates a separate kernel fault injector. The tester can design a fault injection campaign by a configuration file that the fault injection manager refers to. After finishing a fault injection campaign, an output file is created. An output file contains fault injection time, changed values by fault injection, error information, and some debugging data.

The kernel fault injector interacts with KGDB and the target machine console through serial communication. First, it triggers KGDBOC by sending console commands via serial communication. Then, it executes a GDB application on a host machine and connects it to KGDB on the target Linux with GDB Machine Interface(GDB/MI) commands. To insert faults, it either adds breakpoints through GDB or executes a hardware breakpoint controller to enable predefined hardware breakpoints for fault injection through the console. All actions except connection/disconnection to KGDB and hardware breakpoint insertion/deletion are performed by GDB/MI commands. Note that a KGDB breakpoint is used not only for inserting breakpoints but also for monitoring specific code paths or detecting kernel error logs.

The hardware breakpoint controller is a user application that runs on the target machine. It supports both custom and generic hardware breakpoint fault injection by receiving arguments from the host machine and setting appropriate options in the hardware breakpoint fault injection module.

The hardware breakpoint fault injection module is located inside the kernel as a separate module. It handles requests from the hardware breakpoint controller by organizing a proper fault injection campaign which is performed within the exception handler. The

module has responsibility for exception handler registration and hardware breakpoint management.

To use the GDB-based application fault injector, the target application must be executed with a gdbserver. A gdbserver is a program that enables a debugging on the remote application, and it requires a TCP or serial communication. After launching the target application via a gdbserver, the GDB-based application fault injector executes a local GDB and accesses to a gdbserver for fault injection. The rest of the process is similar to KGDB-based fault injection.

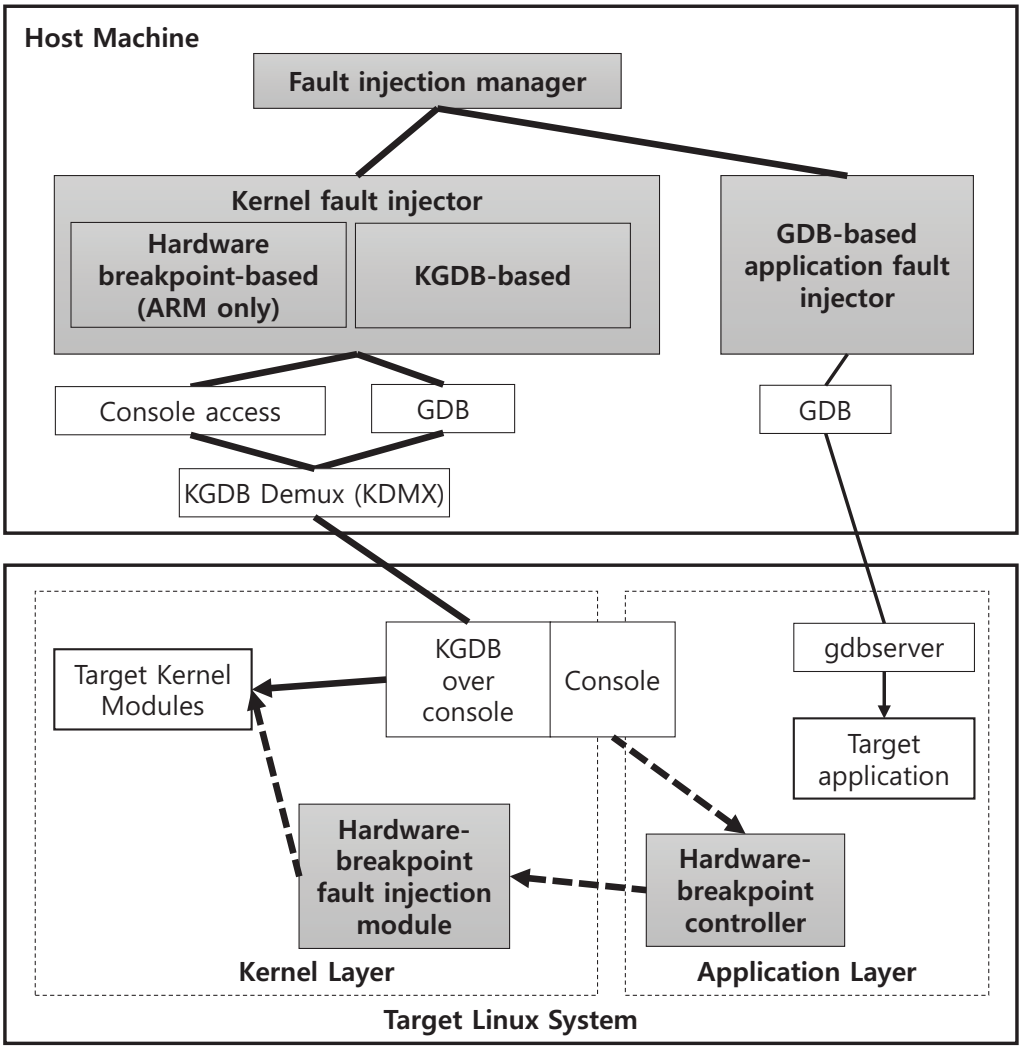


Figure 5.4: The architecture of the fault injection framework

5.2.3 Fault Injection Techniques

To inject faults via KGDB or hardware breakpoints, there are several attributes to be defined. We first explain how these attributes are defined in the proposed fault injection techniques and next how faults are injected in each technique in sequence.

5.2.3.1 Fault Injection Attributes

There are three main attributes for a fault injection campaign.

- *Fault Injection Point*: A fault injection point represents a fault event that is described by a pair of information, position and object. From the hardware perspective, fault time indicates the position where a fault is injected, and a fault object indicates the cause of a fault on a hardware component. In the framework, the fault position is defined by a combination of the source code file and a line number to inject the fault when a specific instruction/code path is executed. The fault object is defined by a specific variable or register that the fault has an effect on. By adjusting these two parameters, the tool defines a fault injection point.

- *Fault Injection Type*: A fault injection type defines how data or state is changed. The most popular fault injection type is bit-flip which inverts a specific bit of data. Depending on a fault injection type, different errors can be emulated for the same fault injection point.

- *Fault Frequency*: Fault frequency indicates how often a fault occurs. As presented in [87], fault frequency is divided into three types: transient, intermittent, and permanent. A transient fault occurs only once, an intermittent fault takes place repeatedly, and a permanent fault preserves a fault value permanently.

5.2.3.2 Fault Injection via KGDB

Defining a fault injection point via KGDB is easy because GDB already has a feature to utilize symbol information to trace the execution of a program. With symbol information, the KGDB method can set the fault injection point to any line of the source code by setting a breakpoint at that line and inject a fault by manipulating variable values in the line. In addition, since GDB supports conditional break, we can define a fault injection point conditionally. With a conditional breakpoint, we can narrow down the fault condition in repeated experiments to trace the specific fault case further. Note that not all features provided by GDB are supported by KGDB.

After reaching a fault injection point, we need to change the data to emulate a fault. Since KGDB supports data modification, we can perform data manipulation on variables or registers in various ways, including bit inversion, bit stuck-at-one, and bit stuck-at-zero. Stuck-at-one or stuck-at-zero means that the bit value is fixed to one or zero regardless of the program behavior.

For KGDB fault injection, only transient or long-interval intermittent faults are appropriate fault frequency because of its significant injection overhead. The overhead is mainly caused by a long delay between GDB and KGDB communication. For manipulating data, multiple times of serial communication are needed to transfer multiple GDB commands and results. Since the delay easily becomes hundreds of milliseconds, frequent fault injections should be avoided in the KGDB method. Implementing a transient fault with a KGDB breakpoint is simple because the only job to do is disabling a GDB breakpoint after the first breakpoint hits. The framework provides options to configure the number of faults to occur so that a user can conduct a transient or long-interval intermittent fault in the KGDB method.

5.2.3.3 Fault Injection via Hardware Breakpoint

Unlike the KGDB method that can use the debugging symbol information, the fault injection point for the hardware breakpoint method is restricted to the start of a function. By computing the relative distance from the start position of a function, we can access a function argument or a variable that is placed in a fixed distance away. By manipulating the data value of the variable, we emulate a faulty behavior of the function.

In addition to manipulating the function argument, the hardware breakpoint method can insert some time delay in the function when a breakpoint is hit, without delaying the whole system. Also, by exploiting the fact that a fault is injected at the exception handler, we are able to define a customized fault scenario by writing an appropriate code inside the exception handler. Detailed examples will be discussed in section 7.

Because the fault injection overhead of hardware breakpoint methods is very low compared to KGDB, faults can be injected frequently. Thus, short-interval intermittent or permanent faults can be modeled.

5.2.3.4 Comparison of Two Fault Injection Techniques

We summarize the difference between two complementary methods in Table 5.1, combining both generic and custom hardware breakpoint methods into one. To emulate a transient fault and a long-intermittent fault, the KGDB method is superior to the hardware breakpoint method since it provides rich capabilities to monitor the system status and to set a virtually unlimited number of breakpoints at once. Since it requires no kernel modification, our framework can be installed on a real system with minimal efforts.

On the other hand, to emulate intermittent or permanent faults, the hardware breakpoint method is preferred because of its low injection overhead. For more details about the injection overhead between two techniques, section 5.3.2 presents experimental results on the actual execution time overhead per a single fault injection in both methods. Unlike the KGDB method, the hardware breakpoint method is not intrusive, meaning that

Table 5.1: Comparison of two fault injection techniques

Property	KGDB	Hardware breakpoint
Data-manipulated fault injection	Supported	Supported
Customized fault injection	Not supported	Supported
System during injecting a fault	stop	non-stop
Injection overhead	High	Low
Kernel-recompile	Not needed	Needed for adding new customized fault injection
Breakpoint limitation	1,000	6

other processors continue working while the processor of interest stops at the breakpoint. While the KGDB method supports a single fault model, bit-flip type of data manipulation, the hardware breakpoint method provides other types of fault, time delay, and device failure. To take advantage of both techniques synergistically, we integrate two fault injection techniques into a unified fault injection framework, sharing the same fault injection manager and kernel fault injector modules.

5.2.4 Implementation

In this section, we explain the implementation details of the proposed fault injection framework. We first explain how a fault injection campaign is set up by defining a configuration file that is read by the fault injection manager. Next, the fault experiment workflow is presented based on the configuration file. Finally, error detection mechanisms are presented.

5.2.4.1 Fault Injection Configuration

As explained in the previous section, a fault injection experiment can be configured by the configuration file that the fault injection manager refers to. We use the same file format as the *libconfig* library, C/C++ configuration library, assumes. An example segment of the configuration file is shown in Fig. 5.5. Since the manager can execute multiple experiments concurrently, experiments are defined as an array with the *experiments* key. Each element specifies an experiment with the timing information when and how long the experiment will be performed based on the host machine's wall clock time; *fault_start_time* and *fault_end_time* indicates when to start and finish fault injection campaigns, and *max_monitoring_time* implies the maximum time to monitor the kernel.

Each experiment is associated with a section that defines the experiment method. For example, in Figure 5.5, *experiments* has one element, and its *fault_injector* value is set to *kernel_fault1* which points to a section named *kernel_fault1*. To design an additional experiment, we should add a new experiment element in *experiments* and specify a new section such as *kernel_fault2*. The *kernel_fault1* section contains the necessary information to perform a fault injection campaign. A configuration file also includes other global information that is common to all experiments such as output file path, log directory path, or log printing level.

Depending on which kind of fault injection methods is used, fault configuration is described differently. For KGDB fault injection, we can specify multiple fault injec-

```
...
summary_file_path = "my_report_file";

log_dir = "log_hwb";
log_level = 4;

kernel_fault1: {
        usb_path = "/dev/ttyUSB0";
        gdb_path = "/opt/tools/bin/arm@-@eabi@-@gdb";
        ...
        fault_scenario = ({
                area = "drivers/.../mfc/s5p_mfc_cmd_v6.c";
                variable = "cmd";
                line_number = 29;
                bit_flip_location = 30;
                bit_flip_style = "bit@-@flip";
                condition = "cmd == 4";
                fault_occurrence = "once";
                }, {
        type = "fault_hwb";
                fault_type = "delay";
                area = "drivers/input/evdev.c";
                line_number = 123;
                fault_occurrence = "once";
                timing_delay = 10;
    });

        kernel_log: {
        ...
        };
}

experiments = ({
        experiment_name = "mfc_fault_injector";
        fault_start_time = 2000;
        fault_end_time = 15000;
        max_monitoring_time = 40000;
        fault_injector = "kernel_fault1";
});
...
```

Figure 5.5: An example segment of the kernel fault injector configuration file

tion points in the *fault_scenario* field. Each item of *fault_scenario* includes the location information of the source code file, line number, variable name, and break condition. Moreover, the fault injection style can be configured further to specify fault injection frequency, bit-flip location, and bit-flip style. Specification of generic hardware breakpoint fault injection is similar, but it has some different options. It needs to specify *fault_type* to indicate which kind of faults is used. Each fault type is associated with particular options. In Figure 5.5, *timing_delay* option is specified associated with the "delay" fault type. Custom hardware breakpoint fault injection has no *fault_scenario* field. It is defined by a special section because most options are not shared with other fault injection methods.

The configuration file additionally supports auxiliary options such as watching the specific variable or kernel logs. To collect error logs from the kernel, a breakpoint can be set at a specific location, and this information is defined in the *kernel_log* section. The *kernel_log* section has options for setting a source code and line number. Since different kernel versions may have different kernel codes for printing kernel logs, configurable options are provided to support different kernel versions. For supporting a trace to error, the call stack and the current process information are collected as a log. In Figure 5.5, the *fault_scenario* defines two fault injection points and one *kernel_log* section whose details are omitted due to space limitation.

5.2.4.2 Fault Experiment Workflow

After receiving the configuration data from the fault injection manager, a kernel fault injector manages each fault experiment according to the workflow, as described in Figure 5.6. First of all, the kernel fault injector executes GDB and establishes serial communication with a console on the target machine. Then, the kernel fault injector sets up a connection to KGDB through GDB and inserts KGDB breakpoints. Since the hardware breakpoint controller is a user application running on the target machine and inserts a

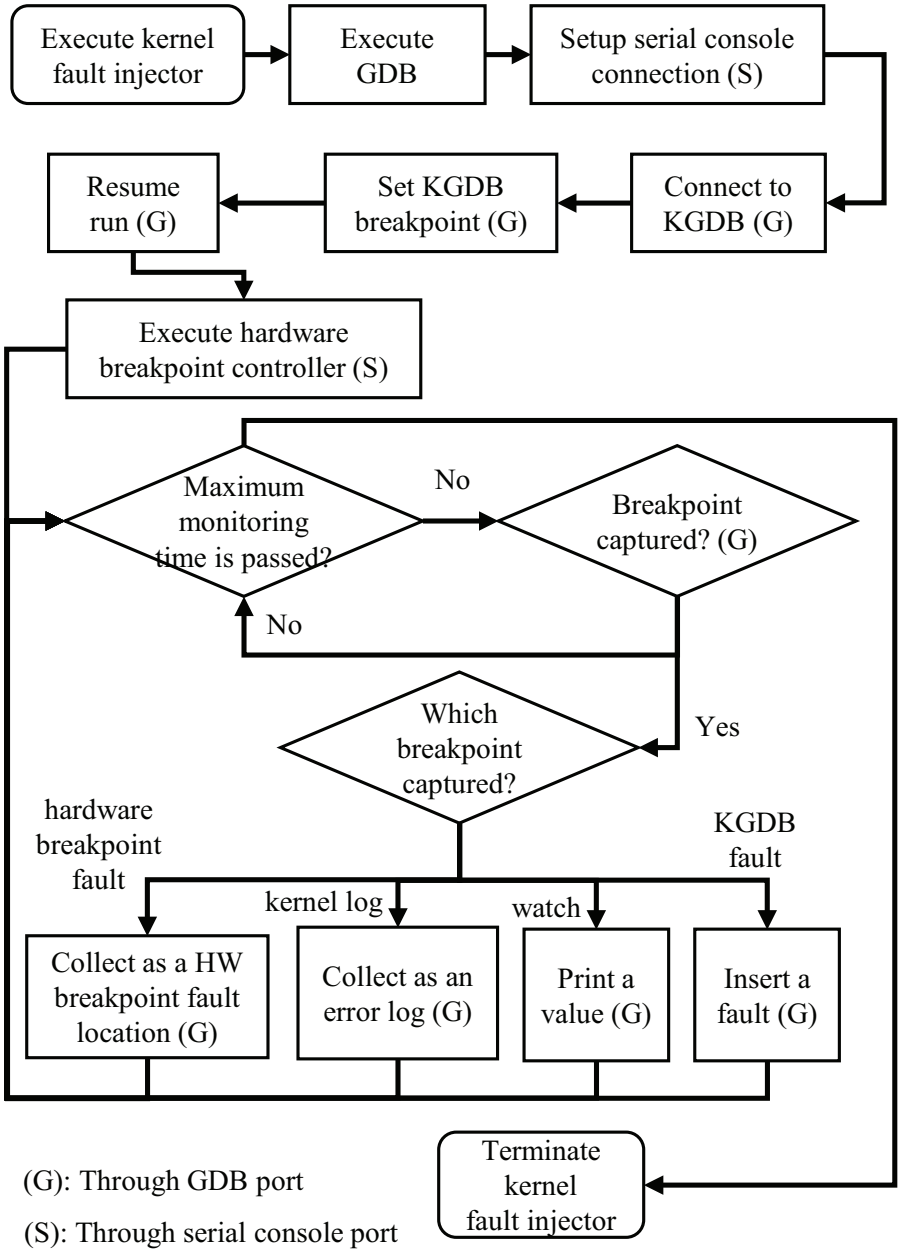


Figure 5.6: The workflow of fault injection framework

breakpoint with a system call, the kernel must be resumed by KGDB.

Until the specified maximum monitoring time is reached, the target system is monitored by KGDB. During the experiment, manual workload generation such as running an app or pressing input devices is needed depending on the location of faults to be injected. If a breakpoint is captured, it is handled differently, depending on the following four cases: a KGDB breakpoint, a hardware breakpoint, a kernel error log, a variable to be watched. When the maximum monitoring time is passed or kernel panic occurs, the kernel fault injector collects the final result and is terminated by the fault injection manager.

To conduct experiments with KGDB, serial port communication with the KGDB module is implemented via GDB/MI which is an interface through a pipelined interprocess communication between GDB and a third-party tool. Some of the major commands used by the kernel fault injector are listed in Table 5.2. For controlling KGDB breakpoints, *-break-insert*, *-break-condition*, and *-break-delete* commands are used. For data watch and manipulation, *-var-assign* and *-data-evaluate-expression* are used. Command *-symbol-list-lines* is used for hardware breakpoints because the hardware breakpoint controller needs a virtual address to set a breakpoint.

For hardware breakpoint fault injection, we have implemented the hardware breakpoint module and the hardware breakpoint fault injection controller. Setting and clearing hardware breakpoints and customizing exception handlers to run fault injection functions must be done at the kernel level. Therefore, the hardware breakpoint module which is responsible for these operations must be implemented as a kernel module. This kernel module resides in the target kernel, and it receives *ioctl* commands from the user level program. The hardware breakpoint controller is a user-level program that serves as a bridge between the KGDB module and the hardware breakpoint module.

The current implementation of our proposed fault injection framework contains some custom hardware breakpoints on specific hardware interface modules to emulate

Table 5.2: The main GDB/MI commands used for fault injection via KGDB

GDB/MI commands	Description	Usage
-break-insert	Insert breakpoint	Insert fault injection point
-break-condition	Add condition on breakpoint	Insert fault injection point with condition
-break-delete	Delete breakpoint	Remove injection point for transient fault
-var-assign	Set variable	Update manipulated variable
-data-evaluate-expression	Check variable value	Get variable, process information, manipulated value
-symbol-list-lines	List line and address pair of specific source file	Convert source line number to virtual address

non-trivial errors such as device failures and timing delay. Faults are injected in the exception handler. At the time when the breakpoint event arises, we are able to access the register values of the handler. By manipulating those values, we can change not only the values of function arguments but also the resuming point after finishing the exception handler routine. This way, non-trivial errors such as device failure and no response could be emulated.

5.2.4.3 Error Detection Mechanism

KGDB can detect some kinds of errors occurred in the target Linux. The fault injector supports three kinds of error detection: kernel panic, hang, and kernel error log. Panic detection is naturally supported because KGDB notices a panic to remote GDB with signal information such as segmentation fault.

Kernel error logs are printed when the kernel or a device driver uses *printk* or a similar log-printing function. A breakpoint is inserted at *vprintk_emit* function which stores error logs to buffer. Each kernel log has a degree, and not all kernel logs mean an error. To collect appropriate error logs, the minimum log level can be adjusted in the fault configuration file. Note that kernel log capturing from KGDB incurs significant overhead because of using a KGDB breakpoint. To avoid the overhead, the fault injector can also

collect logs which are printed on a serial console of the target machine.

In case the target Linux does not respond to both a serial console and a GDB command, hanging error is suspected. To detect hanging, the kernel fault injector sends a special command which disables the console on Linux and invokes an interrupt to communicate with KGDB. It is similar to sending a SIGINT signal by pressing Ctrl+Z in a GDB application. If Linux responds nothing to this command, the kernel fault injector decides that Linux hangs.

5.2.4.4 Tool Usage

Our tool works on a command-line interface, and a shell script is used to perform fault injection campaign. At first, the script executes the Kdmx module to create two pseudo-terminal serial device paths and writes the serial device paths to the fault injection configuration file. Next, the script starts the fault injection manager with the configuration file as an argument. The fault injection manager reads the configuration file and runs the kernel fault injector to perform fault injection experiments. Because our tool does not support automatic workload generation, the user must generate the workload manually during the fault injection experiment. After finishing the experiment, the tool produces console output logs and the summarized result of the fault injection test.

Table 5.3: Hardware and software specification of ODROID-XU4

Feature	Description
Processor	Samsung Exynos5 Octa (ARM Cortex-A15 Quad 2Ghz, ARM Cortex-A7 Quad 1.3GHz)
Memory	2Gbyte LPDDR3 RAM at 933MHz
OS	Android 4.4.4
Kernel	Linux Kernel 3.10.9

5.3 Experiments

5.3.1 Experiment Setup

To perform fault injection experiments, our tool is implemented on an ODROID-XU4 system which has the following specifications as shown in Table 5.3. The Linux kernel for ODROID-XU4 is distributed from the manufacturer's GitHub website[1]. The host machine used in the experiments is an i7-3770 system with 8GB memory, running Ubuntu Linux 14.04. The host machine and ODROID-XU4 are connected through a USB-UART bridge cable that is plugged into the ODROID serial console port. To use KGDB, we need to change the kernel configuration to enable the KGDB option that can be found in *menuconfig*. Except for changing the configuration, no kernel source modification is required. For ARM hardware breakpoints, we added an additional kernel module, hardware breakpoint fault injection module, to the kernel.

We first examine the performance overhead caused by two fault injection techniques. Next, we present some fault injection experiments that demonstrate the fault injection capabilities of our proposed framework. The observed events are summarized in Table 5.5.

5.3.2 Performance Comparison of Two Fault Injection Methods

Fault injection overhead is defined by the additional execution time caused by fault injection operation and its resultant behavior. To measure the injection overhead of both

[1] https://github.com/hardkernel/linux (branch: odroidxu3-3.10.y-android)

methods, a test module is added in the kernel which handles the *ioctl* operation of a specific device file called */dev/testtarget*. When calling the *ioctl* operation, the test module invokes a dummy function named *function_call_test*. Since the hardware breakpoint method can specify a breakpoint at the starting address of a function, the dummy function is defined for setting a breakpoint. We measured the time duration between the function call of *function_call_test* and the first instruction of the function. A kernel time measurement function with a microsecond time unit, *do_gettimeofday*, is used to acquire the current time at each location. Without any breakpoints, the measured time duration becomes 0 or 1 microsecond which means the execution time is less than a unit of time and is short enough to be ignored.

For hardware breakpoint fault injection, three different versions of the test code have been performed: defining a hardware breakpoint only, a hardware breakpoint with fault injection, and a hardware breakpoint with kernel log printing. Kernel log printing is used for displaying the fault injection result using the *printk* function. Similarly, three different experiments are conducted for KGDB fault injection: setting a breakpoint only, setting a breakpoint and inserting a bit-flip fault, and performing information retrievals such as stack trace and the current process information at a breakpoint. From these experiments, we can divide the fault injection overhead into three parts: setting a breakpoint, injecting a fault, and getting the result back.

The performance measurement results with a single fault injection are shown in Table 5.4. Each result is obtained by taking the average time after repeating the same experiment ten times. Table 5.4 shows that hardware breakpoint fault injection is 38,670 times faster than KGDB fault injection if no kernel log is printed out. If the kernel log is printed in the hardware breakpoint method, the overhead is increased significantly by 1,460 times, which is still 30 times faster than the KGDB method. In the KGDB method, the injection overhead varies depending on how many behaviors are performed during a breakpoint. Because information retrieval needs more communications to obtain

Table 5.4: Single fault injection overhead of KGDB and hardware breakpoint methods

Breakpoint overhead type	Average time (μs)
Hardware breakpoint only	4.00
Hardware breakpoint with bit-flip fault injection	4.67
Hardware breakpoint with kernel log printing	6245.33
KGDB breakpoint only	142277.22
KGDB breakpoint with bit-flip fault injection	180587.67
KGDB breakpoint with full information retrieval	367601.78

debugging information, the overhead is two times higher than the overhead of the KGDB breakpoint fault injection.

The overhead per each breakpoint of two fault injection methods helps us to estimate the total performance penalty during fault injection experiments. The performance penalty depends on the number of breakpoints and the fault injection technique. These experiments confirm that hardware breakpoint is suitable to inject a fault into a frequently-called routine because of low fault injection overhead if kernel log printing is suppressed except when an error is actually detected. Although the breakpoint overhead of KGDB is very high, KGDB is still affordable to test a transient fault and has an advantage of performing a fault injection campaign without any additional kernel module or recompilation.

5.3.3 Bit-flip Fault Experiments

As the first experiment of KGDB fault injection, a bit-flip fault model is tested with a USB keyboard connected with ODROID-XU4 and a Multi Format Codec (MFC) which is a video encoding/decoding IP embedded in the Exynos 5422 application processor.

As for the keyboard, we injected a transient bit-flip fault into the event character device driver. We added a condition to inject a fault to a keyboard input only because the target device driver can be accessed by not only a keyboard input but also a mouse or touchscreen input. We pressed the real keyboard manually during the experiment since software input such as *adb input* is not recognized by the driver. After a fault is injected,

Table 5.5: Observed events according to fault models

Fault model	Method	Workload	Observed events
Bit-flip (Keyboard)	KGDB fault injection	Press 'A' key one time	Endless 'S' key printing
Bit-flip (Multi format codec)	KGDB fault injection	Video play	Kernel panic
Communication delay from eMMC controller	Generic hardware breakpoint	App launch	Delay on an app
Erroneous response from eMMC controller	Custom hardware breakpoint	None	Kernel panic
No response from eMMCcontroller	Custom hardware breakpoint	None	Screen freeze

'*s*' key is printed on a screen endlessly even though '*a*' key is manually pressed once. The reason for this behavior is that after bit-flip changes the keyboard value '*a*' to '*s*', the driver receives a '*a*'-key release event but does not receive a '*s*'-key release event, which causes endless printing of 's' on the screen.

For a KGDB fault injection experiment with MFC, we inserted a transient bit-flip fault into the Exynos MFC driver's register which is used for controlling the MFC. When the MFC is initialized, it calls *CH_INIT_BUFS* which commands frame buffer initialization of the MFC. We replaced the command by bit-flipping to *H2R_CMD_CLOSE_INSTANCE* which means closing an MFC instance. To activate a fault, we executed a video player called *Dice Player* and played a sample video. After clicking the video file, the target machine printed a kernel error log printing '*Abnormal h/w state*' and resulted in a kernel panic.

These experiments show that KGDB fault injection can readily narrow down a fault case to find out a specific condition that causes a fault with no kernel recompilation. We can use the bit flip fault model to emulate the hardware failure if we know which data is affected by the hardware failure.

5.3.4 eMMC Controller Fault Experiments

Three experiments are designed to emulate hardware failure by injecting faults with the hardware breakpoint method. The target hardware is the eMMC controller on the board which serves as an interface between the Linux kernel and the eMMC storage card. The Linux kernel has a device driver module for the eMMC controller. The device driver module maintains a kernel thread called *mmcqd* that is responsible for giving a command to the eMMC controller. We set breakpoints inside this module, and when a breakpoint is hit, we inject a fault into the module to emulate a hardware failure in the eMMC controller.

5.3.4.1 Emulating communication delay between the kernel and the eMMC controller

To write data to the eMMC controller, *mmcqd* gives a write command to the eMMC controller and sleeps, waiting for an interrupt signal from the eMMC controller. After serving the request, the eMMC controller sends an interrupt signal which wakes up *mmcqd*.

To emulate communication delay between the kernel and the eMMC controller, we set a breakpoint using the generic hardware breakpoint method. We can choose a location from the function call stack that stores the call tree established when *mmcqd* gives some commands to the eMMC controller. Otherwise, we may decide a location from the function call stack that stores the call tree established when the eMMC controller wakes up *mmcqd* by an interrupt signal.

We ran the *gallery* Android application to see the effect of the delay. We took screenshots, started the gallery application, browsed the screenshots, and erased them. We gave a delay up to 5 seconds every time the hardware breakpoint is hit. We observed application-level delay such as delay in taking, browsing, and erasing screenshots. There was no other abnormal behavior observed in the system. After clearing the breakpoint,

the system worked as usual.

5.3.4.2 Emulating erroneous response from the eMMC controller

After receiving a response from the eMMC controller, *mmcqd* checks for error in the response at function *mmc_blk_err_check*. We set a breakpoint at the start of this function and changed the response state from non-error to error. The default behavior of *mmcqd* when this kind of error is detected is to retry the command several times and then abort if the erroneous response is detected over and over again. We were able to observe this default behavior in this experiment. As the number of aborted commands to the eMMC controller accumulates, the kernel prints a panic message, and the system goes down.

We used the custom hardware breakpoint method in this experiment. The function that checks for error in the response receives the response data as an argument. In the exception handler, we could get the address of this response data and corrupt the data by writing a custom fault injecting code.

5.3.4.3 Emulating no response from the eMMC controller

We emulated the case where the eMMC controller gives no response to *mmcqd*, by preventing the Linux kernel from waking up *mmcqd* after receiving an interrupt signal from the eMMC controller. We set a breakpoint at function *mmc_request_done* which is responsible for waking up *mmcqd*. The typical behavior after the breakpoint exception handler finishes is that the kernel resumes from the instruction where the breakpoint was hit. However, inside this breakpoint handler, we changed the resuming point of the kernel so that the kernel skips the instruction that wakes up *mmcqd*.

We used the custom hardware breakpoint method again in this experiment. After a fault is injected, there is no way that the kernel can wake up *mmcqd*. Thus any program that needs access to eMMC card (e.g., *cp* command in the console, taking a screenshot)

cannot be executed successfully. Also, the screen froze, and mouse clicking did not work. However, we could observe that programs that do not access the eMMC card work normally. For instance, *ls* and *cd* commands conducted in the console, reading values that are not in the eMMC card such as values in Linux */proc* directory, and the mouse point movement could be executed as usual.

Chapter 6

Conclusion

In this thesis, we present a model-based software development method for parallel and distributed embedded systems. An application is specified by a hierarchical task graph, and two types of atomic tasks are distinguished: control task and computation task. The control task specifies the control behavior of the application, based on the FSM model. Computation tasks follow the dataflow model that explicitly reveals the parallelism of the application that can be exploited easily by changing the mapping of tasks onto processors. By clustering the dataflow tasks that have the fixed number of input and output samples per invocation, we can perform static analysis for each subgraph of the clustered dataflow tasks. Through static analysis, we can check the possibility of deadlock and buffer overflow. After a mapping decision is made, we automatically synthesize the program that will run on each processing element. The proposed program synthesis method has two features that make it distinguished from the related work. First, the proposed technique for communication code synthesis is extensible and flexible to support various communication methods between devices. Second, the fault-tolerant method can be defined at the task level, and fault-tolerant code synthesis is realized by modifying the task graph in the proposed method. In addition, a fault injection tool is used for supporting various fault scenarios, and it is used for verifying the fault tolerance of generated codes.

The proposed software development flow is verified with a real-life surveillance application that runs on four devices that have six processing elements in total. Also, experiments are conducted to examine the trade-off between the execution time and the degree of fault tolerance with various fault-tolerant configurations. In the future, we will support semi-constrained targets to generate applications running on various real-time operating systems. Also, we will generate extra codes in consideration of security as well as fault tolerance. While task graphs can be drawn inside each device, extended models cannot be controlled among devices. Supporting inter-device dynamism is another challenging work to increase a variety of implementation and fault tolerance.

Bibliography

[1] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. Autosar–a worldwide standard is on the road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, volume 62, page 5, 2009.

[2] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[3] Soonhoi Ha and EunJin Jeong. *Embedded Software Design Methodology Based on Formal Models of Computation*, pages 306–325. Springer International Publishing, Cham, 2018.

[4] Soonhoi Ha and Hanwoong Jung. *HOPES: Programming Platform Approach for Embedded Systems Design*, pages 1–31. Springer Netherlands, Dordrecht, 2017.

[5] Gang Zhou, Man-Kit Leung, and Edward A. Lee. A code generation framework for actor-oriented models with partial evaluation. In Yann-Hang Lee, Heung-Nam Kim, Jong Kim, Yongwan Park, Laurence T. Yang, and Sung Won Kim, editors, *Embedded Software and Systems*, pages 193–206, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[6] Sundararajan Sriram and Shuvra S Bhattacharyya. *Embedded multiprocessors: Scheduling and synchronization*. CRC press, 2018.

[7] Haojie Fu, Zan Wang, Xiang Chen, and Xiangyu Fan. A systematic survey on automated concurrency bug detection, exposing, avoidance, and fixing techniques. *Software Quality Journal*, 26(3):855–889, September 2018.

[8] C. Buckl, S. Sommer, A. Scholz, A. Knoll, and A. Kemper. Generating a tailored middleware for wireless sensor network applications. In *2008 IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (sutc 2008)*, pages 162–169, June 2008.

[9] A. Scholz, I. Gaponova, S. Sommer, A. Kemper, A. Knoll, C. Buckl, J. Heuer, and A. Schmitt. Efficient communication in control-oriented embedded networks. In *2009 IEEE Conference on Emerging Technologies Factory Automation*, pages 1–8, Sep. 2009.

[10] H. Evrard and F. Lang. Automatic distributed code generation from formal models of asynchronous concurrent processes. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 459–466, March 2015.

[11] A. J. Salman and A. Al-Yasiri. Sennet: A programming toolkit to develop wireless sensor network applications. In *2016 8th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–7, Nov 2016.

[12] Aviral Shrivastava and Moslem Didehban. Software approaches for in-time resilience. In *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC '19, pages 197:1–197:4, New York, NY, USA, 2019. ACM.

[13] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. Swift: software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254, March 2005.

[14] G. A. Reis, J. Chang, and D. I. August. Automatic instruction-level software-only recovery. *IEEE Micro*, 27(1):36–47, Jan 2007.

[15] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 244–258, Washington, DC, USA, 2007. IEEE Computer Society.

[16] H. So, M. Didehban, Y. Ko, A. Shrivastava, and K. Lee. Expert: Effective and flexible error protection by redundant multithreading. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 533–538, March 2018.

[17] H. So, M. Didehban, A. Shrivastava, and K. Lee. A software-level redundant multithreading for soft/hard error detection and recovery. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1559–1562, March 2019.

[18] Hyesun Hong, Hyunok Oh, and Soonhoi Ha. Hierarchical dataflow modeling of iterative applications. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 39:1–39:6, New York, NY, USA, 2017. ACM.

[19] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, Sep. 1987.

[20] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, Feb 1996.

[21] Chanik Park, Jaewoong Chung, and Soonhoi Ha. Extended synchronous dataflow for efficient dsp system prototyping. In *Proceedings Tenth IEEE International Workshop on Rapid System Prototyping. Shortening the Path from Specification to Prototype (Cat. No.PR00246)*, pages 196–201, June 1999.

[22] Hyunok Oh and Soonhoi Ha. Fractional rate dataflow model and efficient code synthesis for multimedia applications. *SIGPLAN Not.*, 37(7):12–17, June 2002.

[23] Bishnupriya Bhattacharya and Shuvra S Bhattacharyya. Parameterized dataflow modeling for dsp systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, 2001.

[24] S. Stuijk, M. Geilen, B. Theelen, and T. Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *2011 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*, pages 404–411, July 2011.

[25] Hanwoong Jung, Chanhee Lee, Shin-Haeng Kang, Sungchan Kim, Hyunok Oh, and Soonhoi Ha. Dynamic behavior specification and dynamic mapping for real-time embedded systems: Hopes approach. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):135, 2014.

[26] Hanwoong Jung, Hyunok Oh, and Soonhoi Ha. Multiprocessor scheduling of a multi-mode dataflow graph considering mode transition delay. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(2):37, 2017.

[27] David MacKenzie, Tom Tromey, and Alexandre Duret-Lutz. Gnu automake. *User Manual, for Automake version*, 1, 1995.

[28] Z. Kalal, K. Mikolajczyk, and J. Matas. Tracking-learning-detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(7):1409–1422, July 2012.

[29] Google. Coral usb accelerator, https://coral.withgoogle.com/products/accelerator/, 2019.

[30] ROBOTIS. Turtlebot3,
http://emanual.robotis.com/docs/en/platform/turtlebot3/overview/, 2019.

[31] ROBOTIS. Opencr 1.0,
http://emanual.robotis.com/docs/en/parts/controller/opencr10/, 2019.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[33] NVIDIA. Jetson agx xavier.
https://developer.nvidia.com/embedded/buy/jetson-agx-xavier-devkit, 2019.

[34] Dimitri Van Heesch. Doxygen: Source code documentation generator tool. *URL: http://www. doxygen. org*, 2008.

[35] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphviz—open source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer, 2001.

[36] Shuvra S. Battacharyya, Edward A. Lee, and Praveen K. Murthy. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, Norwell, MA, USA, 1996.

[37] Jeffrey J Tsay. *A Code Generation Framework for Ptolemy II.* Citeseer, 2000.

[38] Stavros Tripakis, Dai Bui, Marc Geilen, Bert Rodiers, and Edward A. Lee. Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs. *ACM Trans. Embed. Comput. Syst.*, 12(3):83:1–83:26, April 2013.

[39] Maher Fakih and Sebastian Warsitz. Automatic sdf-based code generation from simulink models for embedded software development. In *International Workshop on High Performance Energy Efficient Embedded Systems*, 01 2017.

[40] Gustav Cedersjö and Jörn W. Janneck. Software code generation for dynamic dataflow programs. In *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '14, pages 31–39, New York, NY, USA, 2014. ACM.

[41] Object Management Group. Semantics of a foundational subset for executable uml models, dec 2018.

[42] Object Management Group. Action language for foundational uml, jun 2017.

[43] Federico Ciccozzi, Antonio Cicchetti, and Mikael Sjödin. On the generation of full-fledged code from uml profiles and alf for complex systems. In *2015 12th International Conference on Information Technology-New Generations*, pages 81–88. IEEE, 2015.

[44] Federico Ciccozzi. On the automated translational execution of the action language for foundational uml. *Software & Systems Modeling*, 17(4):1311–1337, 2018.

[45] Asma Charfi, Chokri Mraidha, and Pierre Boulet. An optimized compilation of uml state machines. In *2012 IEEE 15th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 172–179. IEEE, 2012.

[46] Object Management Group. Omg system modeling language, dec 2019.

[47] Andrea Enrici, Ludovic Apvrille, and Renaud Pacalet. A model-driven engineering methodology to design parallel and distributed embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 22(2):34, 2017.

[48] Mohammad Hossein, Askari Hemmat, Otmane Ait Mohamed, and Mounir Boukadoum. Towards code generation for arm cortex-m mcus from sysml activity diagrams. In *2016 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 970–973. IEEE, 2016.

[49] Mario Bambagini and Marco Di Natale. A code generation framework for distributed real-time embedded systems. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, pages 1–10. IEEE, 2012.

[50] Eclipse Foundation. Accelo, dec 2019.

[51] Object Management Group. Uml profile for marte, apr 2019.

[52] Federico Ciccozzi, Tiberiu Seceleanu, Diarmuid Corcoran, and Detlef Scholle. Uml-based development of embedded real-time software on multi-core in practice: Lessons learned and future perspectives. *IEEE Access*, 4:6528–6540, 2016.

[53] Héctor Posadas, Pablo Peñil, Alejandro Nicolás, and Eugenio Villar. Automatic synthesis of communication and concurrency for exploring component-based system implementations considering uml channel semantics. *Journal of Systems Architecture*, 61(8):341–360, 2015.

[54] Nicolas Harrand, Franck Fleurey, Brice Morin, and Knut Eilif Husa. Thingml: A language and code generation framework for heterogeneous targets. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, MODELS '16, pages 125–135, New York, NY, USA, 2016. ACM.

[55] B. Morin, N. Harrand, and F. Fleurey. Model-based software engineering to tame the iot jungle. *IEEE Software*, 34(1):30–36, Jan 2017.

[56] Imad Berrouyne, Mehdi Adda, Jean-Marie Mottu, Jean-Claude Royer, and Massimo Tisi. Cypriot: Framework for modelling and controlling network-based iot applications. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, pages 832–841, New York, NY, USA, 2019. ACM.

[57] S. E. V. and P. Samuel. Automatic code generation from uml state chart diagrams. *IEEE Access*, 7:8591–8608, 2019.

[58] Marian Sorin Adam, Morten Larsen, Kjeld Jensen, and Ulrik Pagh Schultz. Rule-based dynamic safety monitoring for mobile robots. *Journal of Software Engineering for Robotics*, 7(1):120–141, 2016.

[59] Sorin Adam, Morten Larsen, Kjeld Jensen, and Ulrik Pagh Schultz. Towards rule-based dynamic safety monitoring for mobile robots. In *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 207–218. Springer, 2014.

[60] Sorin Adam, Marco Kuhrmann, and Ulrik Pagh Schultz. Automatic code generation in practice: experiences with embedded robot controllers. *ACM SIGPLAN Notices*, 52(3):104–108, 2016.

[61] Bruno Morelli, Riccardo Schiavi, Claudio Scordino, Paolo Gai, and Marco Di Natale. Automatic generation of controls code from models for real-time linux platforms. In *15th Real-Time Linux Workshop (RTLWS)*. Citeseer, 2013.

[62] SciLab at INRIA. Scicoslab, dec 2019.

[63] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel Computing*, 22(6):789 – 828, 1996.

[64] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller. Ucx: An open source framework for hpc network apis and beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, pages 40–43, Aug 2015.

[65] Junchul Choi, Hoeseok Yang, and Soonhoi Ha. Optimization of fault-tolerant mixed-criticality multi-core systems with enhanced wcrt analysis. *ACM Trans. Des. Autom. Electron. Syst.*, 24(1):6:1–6:26, December 2018.

[66] E. Jeong, N. Lee, J. Kim, D. Kang, and S. Ha. Fifa: A kernel-level fault injection framework for arm-based embedded linux system. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 23–34, March 2017.

[67] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham. Ferrari: a tool for the validation of system dependability properties. In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 336–344, 1992.

[68] Thomas Naughton, Wesley Bland, Geoffroy Vallee, Christian Engelmann, and Stephen L. Scott. Fault injection framework for system resilience evaluation: Fake faults for finding future failures. In *Proceedings of the 2009 Workshop on Resiliency in High Performance*, Resilience '09, pages 23–28. ACM, 2009.

[69] Moslem Didehban and Aviral Shrivastava. nzdc: A compiler technique for near zero silent data corruption. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2016.

[70] Dmitrii Kuvaiskii, Oleskii Oleksenko, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Elzar: Triple modular redundancy using intel avx (practical experience report). In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 646–653. IEEE, 2016.

[71] Francisco Afonso, Carlos Silva, Nuno Brito, Sergio Montenegro, and Adriano Tavares. Aspect-oriented fault tolerance for real-time embedded systems. In *Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software*, page 2. ACM, 2008.

[72] C. Buckl, D. Sojer, and A. Knoll. Ftos: Model-driven development of fault-tolerant automation systems. In *2010 IEEE 15th Conference on Emerging Technologies Factory Automation (ETFA 2010)*, pages 1–8, Sep. 2010.

[73] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: a methodology and some applications. *Software Engineering, IEEE Transactions on*, 16(2):166–182, 1990.

[74] Henrique Madeira, M찼rio Rela, Francisco Moreira, and Jo찾oGabriel Silva. Rifle: A general purpose pin-level fault injector. In Klaus Echtle, Dieter Hammer, and David Powell, editors, *Dependable Computing - EDCC-1*, volume 852 of *Lecture Notes in Computer Science*, pages 197–216. Springer Berlin Heidelberg, 1994.

[75] Johan Karlsson, Peter Liden, Peter Dahlgren, Rolf Johansson, and Ulf Gunneflo. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE Micro*, 14(1):8–11, 13–23, 1994.

[76] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz. Electromagnetic fault injection: Towards a fault model on a 32-bit microcontroller. In *Fault Diagnosis and Tolerance in Cryptography (FDTC), 2013 Workshop on*, pages 77–88, 2013.

[77] Jr. Samson, J.R., W. Moreno, and F. Falquez. A technique for automated validation of fault tolerant designs using laser fault injection (lfi). In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 162–167, 1998.

[78] Mei-Chen Hsueh, T.K. Tsai, and R.K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[79] D. Skarin, R. Barbosa, and J. Karlsson. Goofi-2: A tool for experimental dependability assessment. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 557–562, 2010.

[80] D.T. Stott, B. Floering, D. Burke, Z. Kalbarczpk, and R.K. Iyer. Nftape: a framework for assessing dependability in distributed systems with lightweight fault injectors. In *Computer Performance and Dependability Symposium, 2000. IPDS 2000. Proceedings. IEEE International*, pages 91–100, 2000.

[81] N.P. Kropp, P.J. Koopman, and D.P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 230–239, 1998.

[82] J.M. Bieman, D. Dreilinger, and Lijun Lin. Using fault injection to increase software test coverage. In *Software Reliability Engineering, 1996. Proceedings., Seventh International Symposium on*, pages 166–174, 1996.

[83] Kai Cong, Li Lei, Zhenkun Yang, and Fei Xie. Automatic fault injection for driver robustness testing. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 361–372. ACM, 2015.

[84] R. Natella, D. Cotroneo, J.A. Duraes, and H.S. Madeira. On fault representativeness of software fault injection. *Software Engineering, IEEE Transactions on*, 39(1):80–96, 2013.

[85] Roberto Natella, Domenico Cotroneo, and Henrique S. Madeira. Assessing dependability with software fault injection: A survey. *ACM Comput. Surv.*, 48(3):44:1–44:55, February 2016.

[86] Adam Lackorzynski. Minicom project. https://alioth.debian.org/projects/minicom/, 2003.

[87] S. Han, K.G. Shin, and H.A. Rosenberg. Doctor: an integrated software fault injection environment for distributed real-time systems. In *Computer Performance and Dependability Symposium, 1995. Proceedings., International*, pages 204–213, 1995.

요 약

소프트웨어 설계 생산성 및 유지보수성을 향상시키기 위해 다양한 소프트웨어 개발 방법론이 제안되었지만, 대부분의 연구는 응용 소프트웨어를 하나의 프로세서에서 동작시키는 데에 초점을 맞추고 있다. 또한, 임베디드 시스템을 개발하는 데에 필요한 지연이나 자원 요구 사항에 대한 비기능적 요구 사항을 고려하지 않고 있기 때문에 일반적인 소프트웨어 개발 방법론을 임베디드 소프트웨어를 개발하는 데에 적용하는 것은 적합하지 않다.

이 논문에서는 병렬 및 분산 임베디드 시스템을 대상으로 하는 소프트웨어를 모델로 표현하고, 이를 소프트웨어 분석이나 개발에 활용하는 개발 방법론을 소개한다. 우리의 모델에서 응용 소프트웨어는 계층적으로 표현할 수 있는 여러 개의 태스크로 이루어져 있으며, 하드웨어 플랫폼과 독립적으로 명세한다. 태스크 간의 통신 및 동기화는 모델이 정의한 규약이 정해져 있고, 이러한 규약을 통해 실제 프로그램을 실행하기 전에 소프트웨어 에러를 정적 분석을 통해 확인할 수 있고, 이는 응용의 검증 복잡도를 줄이는 데에 기여한다. 지정한 하드웨어 플랫폼에서 동작하는 프로그램은 태스크들을 프로세서에 매핑한 이후에 자동적으로 합성할 수 있다.

위의 모델 기반 소프트웨어 개발 방법론에서 사용하는 프로그램 합성기를 본 논문에서 제안하였는데, 명세한 플랫폼 요구 사항을 바탕으로 병렬 및 분산 임베디드 시스템을에서 동작하는 코드를 생성한다. 여러 개의 정형적 모델들을 계층적으로 표현하여 응용의 동적 행태를 나타고, 합성기는 여러 모델로 구성된 계층적인 모델로부터 병렬성을 고려하여 태스크를 실행할 수 있다. 또한, 프로그램 합성기에서 다양한 플랫폼이나 네트워크를 지원할 수 있도록 코드를 관리하는 방법도 보여주고 있다. 본 논문에서 제시하는 소프트웨어 개발 방법론은 6개의 하드웨어 플랫폼과 3 종류의 네트워크로 구성되어 있는 실제 감시 소프트웨어 시스템 응용 예제와 이종 멀티 프로세서를 활용하는 원격 딥 러닝 예제를 수행하여 개발 방법론의 적용 가능성을 시험하였다. 또한, 프로그램 합성기가 새로운 플랫폼이나 네트워크를 지원하기 위해 필요로 하는 개발 비용도 실제 측정 및 예측하여 상대적으로 적은 노력으로 새로운 플랫폼을 지원할 수 있음을 확인하였다.

많은 임베디드 시스템에서 예상치 못한 하드웨어 에러에 대해 결함을 감내하는 것을 필요로 하기 때문에 결함 감내에 대한 코드를 자동으로 생성하는 연구도 진행하였다. 본 기법에서 결함 감내 설정에 따라 태스크 그래프를 수정하는 방식을 활용하였으며, 결함 감내의 비기능적 요구 사항을 응용 개발자가 쉽게 적용할 수 있도록 하였다. 또한, 결함 감내 지원하는 것과 관련하여 실제 수동으로 구현했을 경우와 비교하였고, 결함 주입 도구를 이용하여 결함 발생 시나리오를 재현하거나, 임의로 결함을 주입하는 실험을 수행하였다.

마지막으로 결함 감내를 실험할 때에 활용한 결함 주입 도구는 본 논문의 또 다른 기여 사항 중 하나로 리눅스 환경으로 대상으로 응용 영역 및 커널 영역에 결함을 주입하는 도구를 개발하였다. 시스템의 견고성을 검증하기 위해 결함을 주입하여 결함 시나리오를 재현하는 것은 널리 사용되는 방법으로, 본 논문에서 개발된 결함 주입 도구는 시스템이 동작하는 도중에 재현 가능한 결함을 주입할 수 있는 도구이다. 커널 영역에서의 결함 주입을 위해 두 종류의 결함 주입 방법을 제공하며, 하나는 커널 GNU 디버거를 이용한 방법이고, 다른 하나는 ARM 하드웨어 브레이크포인트를 활용한 방법이다. 응용 영역에서 결함을 주입하기 위해 GDB 기반 결함 주입 방법을 이용하여 동일 시스템 혹은 원격 시스템의 응용에 결함을 주입할 수 있다. 결함 주입 도구에 대한 실험은 ODROID-XU4 보드에서 진행하였다.

주요어 : 코드 생성, 결함 감내, 데이터플로우 모델, 임베디드 소프트웨어 설계 방법론, 결함 주입, 멀티플랫폼 프로그래밍, 네트워크 프로그래밍, 병렬 및 분산 시스템

학번 : 2015-30273